

Hands On AGK BASIC

**A Beginner's Guide to Multi-Platform
Games Programming**

Alistair Stewart

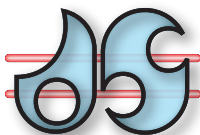


Digital Skills

Hands On AGK BASIC

A Beginner's Guide to Multi-Platform Games Programming

Alistair Stewart



www.digital-skills.co.uk
tel: +44(0)1465 861 638

Digital Skills

Milton
Barr
Girvan
Ayrshire
KA26 9TY
United Kingdom

Copyright © 2012-2013 Alistair Stewart

All rights reserved.

No part of this work may be reproduced or used in any form without the written permission of the author.

Although every effort has been made to ensure accuracy, the author and publisher accept neither liability nor responsibility for any loss or damage arising from the information in this book.

AGK BASIC is produced by The Game Creators Ltd.

Cover Design: Sébastien Leroux

Printed June 2012
Updated February 2013
ebook Updated April 2013

Title: Hands On AGK BASIC

ISBN: 978-1-874107-14-9

Other Titles Available:

Hands On DarkBASIC Pro Vols 1 & 2
Hands On Milkshape

Table of Contents

Foreword	i
Preface	iii
Acknowledgements	iii
How to Get the Most Out of this Book	iv

Chapter 1 - Algorithms

Designing Algorithms	2
Following Instructions	2
Control Structures	3
Sequence	3
Selection	4
Complex Conditions	10
Iteration	14
Data	20
Levels of Detail	22
Checking for Errors	26
Summary	29
Solutions	32

Chapter 2 - Starting AGK

Programming a Computer	36
Introduction	36
The Compilation Process	36
Summary	38
Starting AGK	39
Introduction	39
Starting Up AGK	39
The Program Code	42
Transferring Your App to a Tablet or Smartphone	43
Summary	44
First Statements in AGK BASIC	45
Introduction	45
Print()	45
Adding Comments	47
PrintC()	47
Other Statements which Modify Output	48
Summary	52
The Second Source File	54
A Splash Screen	55
Starting a New Project	56
App Window Properties	57
Measurements	57
Summary	59
Solutions	61

Chapter 3 - Data

Program Data	64
Introduction	64
Constants	64
Variables	64
Named Constants	68
Summary	69
Allocating Values to Variables	70
Introduction	70
The Assignment Statement	70
The Print() Statement Again	77
Acquiring Data	79
User Input	87
Summary	90
Testing Sequential Code	91
Solutions	93

Chapter 4 - Selection

Binary Selection	98
Introduction	98
if	98
The Other if Statement	107
Summary	108
Multi-Way Selection	109
Introduction	109
Nested if Statements	109
The select Statement	112
Testing Selective Code	115
Summary	117
Solutions	118

Chapter 5 - Iteration

Iteration	124
Introduction	124
The while .. endwhile Construct	124
The repeat .. until Construct	126
The for..next Construct	128
Finding the Smallest Value in a List of Values	133
The exit Statement	134
The do .. loop Construct	135
Nested Loops	135
Nested for Loops	136
Testing Iterative Code	137
Summary	139
Solutions	140

Chapter 6 - A First Look at Resources

Resources - A First Look	146
Introduction	146
Images	146
Images in AGK	149
Sound	156
Music	159
Detecting User Interaction	163
Text Resources	165
Later	170
Summary	170
Solutions	172

Chapter 7 - Spot the Difference Game

Game - Spot the Difference	176
Introduction	176
Game Design	176
Game Code	182
Solutions	188

Chapter 8 - User-Defined Functions

Functions	192
Introduction	192
Functions	192
Parameters	196
Summary	206
BASIC Subroutines	207
Introduction	207
Creating a Subroutine	207
A Library of Functions	209
Introduction	209
Creating a Library	209
Creating Modular Software	211
Introduction	211
Top-Down Programming	212
Bottom-Up Programming	219
Structure Diagrams	221
Summary	222
Solutions	224

Chapter 9 - String and Math Functions

String Functions	232
Introduction	232
String-Handling Functions	232
Creating Your Own String Functions	242
Summary	248

Math Functions	250
Introduction	250
Coordinates	250
Trigonometric Functions	251
Other Math Functions	259
Summary	262
Solutions	265

Chapter 10 - Arrays

Arrays	270
Problems with Simple Variables	270
One Dimensional Arrays	271
Using Arrays	276
Dynamic Arrays	293
The undim Statement	294
Multi-dimensional Arrays	294
3-Dimensional Arrays and Higher	295
Arrays and Functions	296
Summary	296
Solutions	297

Chapter 11 - Data Types and Operators

Data Storage	304
Introduction	304
Declaring Variables	304
Type Definitions	305
Summary	310
Data Manipulation	311
Introduction	311
Other Number Systems	311
Shift Operators	312
Bitwise Boolean Operators	314
A Practical Use For Bitwise Operations	317
Summary	318
Solutions	320

Chapter 12 - File Handling

Files	324
Introduction	324
Accessing Files	324
File Management	330
Folder Management	331
Zip Files	335
Summary	336
Solutions	338

Chapter 13 - Particles

Particles	342
Introduction	342
Creating Particles	342
Retrieving Particles Data	355
Summary	359
Solutions	361

Chapter 14 - Text

Text	366
Introduction	366
Review	366
Further Text Statements	367
Text Character Statements	375
Summary	385
Solutions	389

Chapter 15 - User Input

Virtual Buttons	392
Introduction	392
Virtual Button Statements	392
Using Multiple Virtual Buttons	397
Summary	399
Keyboard Input	400
Introduction	400
Text-Input Statements	400
Summary	404
Edit Box Statements	405
Introduction	405
Edit Box Statements	405
Summary	418
Joystick Input	421
Introduction	421
Virtual Joystick Statements	421
Physical Joysticks	427
Summary	430
Device Dependent Input	432
Introduction	432
Accelerometer Statements	432
Mouse Statements	435
Joystick Statements	437
Keyboard Statements	440
Device Identity	442
Summary	442
Solutions	444

Chapter 16 - Images

Images	450
Introduction	450
Review	450
Further Image Statements	450
The ImageJoiner Utility	455
Atlas Texture Files and Proportional Fonts	456
Manipulating Images	457
Image Selection from Storage	460
Using a Device's Camera	461
Mapping Images to Sprites	463
Summary	466
Solutions	468

Chapter 17 - Sprites

Sprites	470
Introduction	470
Review	470
Other Sprite Statements	471
The Sprite Offset Feature	494
Sprite Bounding Areas	499
Sprite Groups	505
Moving Sprites	511
Controlling Speed	523
Ray Casting	524
Summary	532
A Jigsaw Puzzle Game	535
Introduction	535
The Game	535
The Data Files	535
Game Layout	536
The Game Code	537
Solutions	541

Chapter 18 - Animated Sprites

Introduction	550
Using an Animated Sprite	550
A Card Trick	556
Summary	558
An Asteroid Game	560
Introduction	560
Game Layout	560
Game Logic	561
Game Resources	561
Game Code	561
Solutions	573

Chapter 19 - Screen Handling

Screen Handling	580
Introduction	580
Screen-Related Statements	580
Zooming and Scrolling	583
Touch Statements	595
Summary	602
Secrets of Sync()	604
Summary	608
Solutions	609

Chapter 20 - Physics

Sprite Physics - 1	614
Introduction	614
Basic Physic Statements	614
Physics Collisions	628
Physics Sprite Shapes	630
Summary	634
World Physics	636
Introduction	636
General Statements	636
Forces	638
Summary	641
Sprite Physics - 2	643
Contacts	643
Physics Groups and Categories	648
Physics Ray Casting	653
Summary	656
Joints	658
Introduction	658
Joint Statements	658
Summary	683
Solutions	685

Chapter 21 - Accessing a Network

Multiplayer Games	692
Introduction	692
Hardware Requirements	692
The Host and its Clients	692
Multiplayer Statements	693
Summary	716
Multi-Player Tic Tak Toe	718
Introduction	718
Game Logic	718
Program Code	719
HTTP	727
Introduction	727

HTTP Statements	727
Summary	736
Solutions	738

Chapter 22 - Bits and Pieces

Date and Time	748
Introduction	748
Standard Date Statements	748
Unix Date Statements	749
Time Statements	751
Summary	752
QR Coding	753
Introduction	753
QR Code Statements	753
Summary	755
Advertising	756
Introduction	756
Ad Statements	756
Summary	757
Errors	759
Introduction	759
Error Handling Statements	759
Summary	760
Benchmarking	761
Introduction	761
Benchmarking Statements	761
Summary	765
Paused Apps	766
Solutions	769

Chapter 23 - 3D Graphics

Concepts and Terminology	772
Introduction	772
Modelling Ideas and Terminology	776
Summary	783
Creating a First 3D App	786
Introduction	786
Statements	786
User Control of the Camera	790
Summary	792
Object Creation and Modification	793
Creating Primitives	793
Object Appearance	798
Transforming Objects	803
Cameras	816
Introduction	816
Camera-Related Statements	816
Using Camera Commands to Create First Person Perspective	823
Billboarding	828

Summary	829
Lights	831
Introduction	831
Directional Lights	831
Point Lights	833
Object Reflectivity	835
Summary	836
Collisions	
Introduction	837
Ray Cast Statements	837
Summary	855
Other 3D Related Statements	857
Converting Between Screen and 3D Coordinates	857
Sprite and 3D Depth Settings	863
The Depth Buffer	863
Shaders	866
Quaternion Rotation	869
Summary	872
Solutions	873

Chapter 24 - Memory Blocks

Accessing Memory	884
Introduction	884
Memory Block Statements	885
Storing Characters and Strings in a Memory Block	891
Using a Memory Block as an Array	893
Using a Memory Block as a Record Structure	895
Saving Memory Block Data to a File	904
Summary	908
Memory Blocks for Images	910
Introduction	910
Memory Block Image Statements	910
Mapping a Pixel to a Memory Block	912
Modifying an Image's Data	913
Creating Your Own Images from Scratch	914
Summary	916
Creating a Mandelbrot Image	918
Producing the Program	920
Zooming In	927
Shortcomings	930
Solutions	932

Chapter 25 - Drawing

Drawing Statements	940
Drawing a Line	940
Drawing a Dot	940
Drawing a Rectangle	941
Drawing a Circle	943
Drawing an Ellipse	945

Creating a Data Structure for Basic Shapes	947
Summary	952
Drawing a Simple Bezier Curve	953
Introduction	953
Calculating the Curve	953
Creating a Bezier Curve in Real Time	960
Displaying 3D Models in Wireframe	966
Introduction	966
Developing the Program Logic	968
Implementing the Program	969
Solutions	975
Appendix A - ASCII Codes	939
Index	940

Foreword

by Lee Bamber

When I was nine I received my first personal computer, a VIC-20, which was blessed with over 3K of system memory and a maximum palette of 16 colours. From that moment my universe was slightly larger than the amount of memory it takes to store this paragraph of text. In that universe I created lost civilisations, space battles, deep treks into inhospitable lands and dangerous creatures ready leap out from every dark corner. Granted most of it happened in the imagination of the player, but my audience consisted of my parents, my brothers and my uncle who all thought my ‘games’ were amazing.

What was truly amazing was the rate at which the limits of my universe expanded with more memory, more colours, more speed and a bigger audience to play my ‘games’. We went from back-bedroom build-your-own hobby developers to a global industry worth Billions, and it happened so quickly we still have the original founders of this industry working alongside the newest recruits.

Veteran fogies like me can look back and see so much history that when something new comes along, we can almost instantly compare it to five things it strongly resembles from our own fading recollections. We can also identify when something is utterly game-changing, and it usually happens on an epic scale. For me, that moment was when the term ‘apps’ entered the public consciousness. Before then you had software you went out and bought, because you needed software. When the idea of an ‘app’ emerged, it gave ‘software’ a name change and a leviathan marketing budget to spend to the end of time. We are no longer a community of developers who write software, we’re a community that creates solutions to make life better, and its consumers, not developers, who are deciding what those should be.

Here in lies the problem for us poor, overworked developers. We had our plate full just writing software that worked sufficiently for a period of time on one computer. Now we have to create solutions for everyone, where-ever they are, when-ever they want to use it and what-ever they are using as a ‘computer’ at the time. People today want to use their favourite ‘app’ on their home computer, their phone, their TV, in their car and on their fancy new touch tablet, and they want it instantly and constantly up to date. It’s enough to make you cry!

In the best tradition of software developers, whenever we face an emergent system that requires an impossible amount of resources, we simply change the system. Why have ten developers working on ten different systems when you can have one developer working on a single system, and then have a cleverer system translate that work to the other nine automatically. Sounds great in theory, but the practical application produces a number of very oddly shaped solutions indeed.

Now what if you could spin the time machine forward a few years and grab one of the nicer solutions to this problem and then zip back to the present day and start using it? Well it just so happens that I do have a time machine and did just that. It seems, The Game Creators Ltd of 2015 ‘will be’ working with a new piece of software called AGK (App Game Kit) and they ‘will make’ me promise that providing I don’t upset causality, I can take an early copy back with me to 2011 to help them omega test it. Call it a moment of weakness, but I might have put this copy of the product on a website at www.appgamekit.com.

Apparently the break-through with AGK is that you can develop an app on one

system, and it will be instantly compatible with every other system on the planet. I've only managed to get it working on Windows, Mac, MeeGo, iOS, Android and Bada at the moment, but with some more tweaking of their strange alien code I 'will be' assured I can get it to produce all the other platforms present on Earth, even the ones that don't exist yet.

AGK uses the concept of universal commands. That is, each command will perform the same functionality no matter which system it happens to be running on. It is also input agnostic, so if your application requires an input source that does not exist, AGK will virtualise that input data from another piece of hardware present on the device or emulate it through virtual controls. The result is that you can write an 'app' just once, and the resulting program will run on any device present today and any device in the future too.

As developers we have a few decades of history under our belt and can swell with pride on what we have achieved to date. My prediction is that we've just created the world's largest rod for our backs, and now have to finish what we started. The only way forward is to evolve ten pairs of hands through a fortuitous genetic mutation, or find a solution that lets us meet the demands of the next few decades with confidence, a sense of fun and above all, ten fingers!

Lee Bamber
CEO The Game Creators Ltd
2012

Preface

Welcome to the amazing world of the App Game Kit. This is an application that will allow you to create a program that you can design on one machine and run on just about any other platform.

Want to write a game that will run on your phone or your tablet? No problem! Write the application on your regular computer and transfer it to your other devices - it's easy!

Graphics, animation, sound, touch screen, mouse, joystick, keyboard - your app will cope with them all.

Write your apps and sell them online. Some game apps have sold over 5 million copies.

And although AGK stands for App Game Kit, there's no reason why your creation has to be a game. You can easily write educational material, utilities or any number of applications.

Who is this book for? It's for you. It doesn't matter if you're a programming guru or have never written a line of code in your life. This book assumes only a basic knowledge of computers. If you can run an application, copy, paste, delete data, access the internet, type (even with just one finger), and know just a little basic arithmetic then that's all that assumed. Everything else is here. And for the guru there are plenty of hints and tips that I'm sure you will find helpful.

Some books can be very hard going: pages and pages of detail - most of which you forget as soon as you turn to the next page, or when you fall asleep. We do things differently here. No getting bored reading page after page - you'll have a series of activities to carry out that are designed to reinforce what you've read on the page. And unlike most other books that seem to forget about any tasks they have set you, you'll find a full set of answers to the activities at the end of each chapter.

Enjoy your journey through this book.

Acknowledgements

I'd like to thank Lee Bamber, Paul Johnston and Mike Johnson from The Game Creators for all their help and guidance. Also, thanks to John McKay for his patience and forbearance in testing every example included in the book. As usual, Virginia Marshall did her best to rid the book of any grammar or spelling problems.

As always, any errors remaining are entirely my own.

I am always happy to receive any helpful suggestions on how to improve the book or - heaven forbid - details of any errors you've found.

Contact me at alistair@digital-skills.co.uk.

Alistair Stewart June 2012

Second Edition

It's an almost impossible task to write an up-to-date book on a language that changes as rapidly as AGK. Until now we've published updates and extra chapters to extend the original Hands On AGK BASIC to deal with the many changes and additions of AGK version 1, but now, with AGK version 2 and another swathe of additional and updated commands, I've taken this opportunity to revamp the whole book.

The main change is that the publication has now been split into two volumes, with the more advanced topics such as 3D and networking commands moved to the second volume. But I've also taken the opportunity to make minor corrections to retained text and to check that the sample programs run correctly on the latest version of AGK.

As always, please feel free to email me with any useful suggestions or corrections.

Alistair Stewart April 2014

How to Get the Most Out of this Book

Is learning the basics of computer programming difficult? No, but you do have to put in the effort. Despite other publications promising to have you expert in a day, or a week, I'm sure you're smart enough to know that's not going to happen. So, let's get real: you'll learn how to program using AGK if you put in the work, take your time to make sure you understand something before moving on, and practice, practice, practice.

We've tried to keep things interesting by giving you plenty of practical work to do as you journey through this book, but feel free to try out your own projects as well.

The first chapter is the only one in which you won't need your computer since it concentrates on the basic concepts behind all computer programming. You can, if you wish, work on the second chapter at the same time as you read through Chapter 1. That way, you'll be able to start programming right away.

Take your time with each chapter. Make sure you do each of the activities: they are there to give you a deeper understanding as well as to keep you actively involved. Since most activities require you to create a program, the computer will let you know if you've got it right, but you should still take the time to look at the activity's solution given at the end of the chapter. The solution given may differ from your own but it's always of use to see how others tackle the same problem.

Don't be afraid to reread a section or a whole chapter - it's the second or third reading of something new that finally gets the information across to most people.

If you are already a seasoned programmer you will be able to skip through much of the early chapters. If you have programmed in DarkBASIC before, many of the core statements in AGK are identical to that earlier language, but look out for a few subtle differences such as the lack of READ and DATA statements and the method used to initialise arrays.

The Files for the Book

Many of the programming activities (particularly in later chapters) make use of other resources such as images, sounds, and 3D models. You can download the necessary files from

www.digital-skills.co.uk/downloads/AGK2Downloads.zip

1

Algorithms

In this Chapter:

- ☐ Understanding Algorithms
- ☐ Creating Algorithms
- ☐ Control Structures
- ☐ Boolean Expressions
- ☐ Data Types
- ☐ Stepwise Refinement
- ☐ The Need for Testing

Designing Algorithms

Following Instructions

Activity 1.1

Carry out the following set of instructions in your head.

Think of a number between 1 and 10
Multiply that number by 9
Add up the individual digits of this new number
Subtract 5 from this total
Think of the letter at that position in the alphabet
Think of a country in Europe that starts with that letter
Think of a mammal that starts with the second letter of the country's name
Think of the colour of that mammal

Congratulations! You've just become a human computer. You were given a set of instructions which you have carried out (by the way, did you think of the colour grey?).

That's exactly what a computer does. You give it a set of instructions, the machine carries out those instructions, and that is ALL a computer does. If some computers seem to be able to do amazing things, that is only because someone has written an amazingly clever set of instructions. A set of instructions designed to perform some specific task (like that in Activity 1.1) is known as an **algorithm**.

A clear and concise algorithm should have the following characteristics:

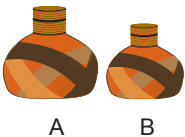
- One instruction per line
- Each instruction is unambiguous
- Each instruction is as brief as possible

Activity 1.2

This time let's see if you can devise your own algorithm.

The task you need to solve is to measure out exactly 4 litres of water. You have two containers. Container A, if filled, will hold exactly 5 litres of water, while container B will hold 3 litres of water. You have an unlimited supply of water and a drain to get rid of any water you no longer need. It is not possible to know how much water is in a container if you only partly fill it from the supply.

If you manage to come up with a solution, see if you can find a second way of measuring out the 4 litres.



As you can see, there are at least two ways to solve the problem given in Activity 1.2. Is one better than the other? Well, if we start by filling container A, the solution needs less instructions, so that might be a good guideline at this point when choosing which algorithm is best.

However, the algorithms that a computer carries out are not written in English like

the instructions shown above, but in a more stylised form using a computer **programming language**. AGK BASIC is one such language. The set of program language instructions which make up each algorithm is then known as a **computer program** or **software**.

Just as we may perform a great diversity of tasks by following different sets of instructions, so the computer can be made to carry out any task for which a program exists.

A traditional disk makes use of a magnetic surface to record information. More recent designs use solid state memory.

Computer programs are normally **copied** (or **loaded**) from a disk into the computer's memory and then **executed** (or **run**). Execution of a program involves the computer performing each instruction in the program one after the other. This it does at impressively high rates, possibly exceeding 160,000 million (or 160 billion) instructions per second (160,000 **mips**).

Depending on the program being run, the computer may act as a word processor, a database, a spreadsheet, a game, a musical instrument or one of many other possibilities. Of course, as a programmer, you are required to design and write computer programs rather than use them. And, more specifically, our programs in this text will be mainly multimedia and game oriented, an area of programming for which AGK has been specifically designed.

Activity 1.3

- a) A set of instructions that performs a specific task is known as what?
- b) What term is used to describe a set of instructions used by a computer?
- c) The speed of a computer is measured in what units?

Control Structures

Although writing algorithms and programming computers can be complicated tasks, there are only a few basic concepts and statements which you need to master before you are ready to start producing software. Luckily, many of these concepts are already familiar to you in everyday situations. If you examine any algorithm, no matter how complex, you will find it consists of only three basic structures:

- **Sequence** where one instruction follows on from another.
- **Selection** where a choice is made between two or more alternative actions.
- **Iteration** where one or more instructions are carried out over and over again.

These structures are explained in detail over the next few pages. All that is needed is to formalise how they are used within an algorithm. This formalisation better matches the structure of a computer program.

Sequence

A set of instructions designed to be carried out one after another, beginning at the first and continuing, without omitting any, until the final instruction is completed, is known as a **sequence**. For example, instructions on how to perform an everyday task such as plant a bush in the garden would be:

Choose spot for planting
Dig hole
Add fertiliser
Place shrub in hole
Refill hole

The set of instructions given earlier in Activity 1.1 is also an example of a sequence.

Activity 1.4

Re-arrange the following instructions to describe how to play a single shot during a golf game:

Swing club forwards, attempting to hit ball
Take up correct stance beside ball
Grip club correctly
Swing club backwards
Choose club

Selection

Binary Selection

Often a group of instructions in an algorithm should be carried out only when certain circumstances arise. For example, if we were playing a simple game with a young child in which we hide a sweet in one hand and allow the child to have the sweet only if she can guess which hand the sweet is in, then we might explain the core idea with an instruction such as

Give the sweet to the child if the child guesses which hand the sweet is in

Notice that when we write a sentence containing the word IF, it consists of two main components:

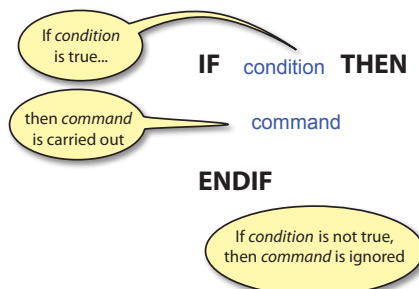
a condition : the child guesses which hand the sweet is in
and
a command : give the sweet to the child

A **condition** (also known as a **Boolean expression**) is a statement that is either true or false in a given situation. The command given in the statement is only carried out if the condition is true at that particular moment and hence this type of instruction is known as an **IF** statement and the command as a **conditional instruction**. Although English would allow us to rewrite the above instruction in many different ways, when we produce a set of formal instructions, as we are required to do when writing algorithms, then we use a specific layout as shown in FIG-1.1, always beginning with the word IF.

FIG-1.1

The IF Statement

Note that there are two alternative actions in this structure: to carry out the command or to ignore it.



Notice that the layout of this instruction makes use of three terms that are always included. These are the words IF, which marks the beginning of the instruction; THEN, which separates the condition from the command; and finally, ENDIF which marks the end of the instruction.

The indentation of the command is important since it helps our eye grasp the structure of our instructions. Appropriate indentation is particularly valuable in aiding readability once an algorithm becomes long and complex. Using this layout, the instruction for our game with the child would be written as:

```
IF the child guesses which hand the sweet is in THEN
    Give the sweet to the child
ENDIF
```

Sometimes, there will be several commands to be carried out when the condition specified is met. For example, in the game of Scrabble we might describe a turn as:

```
IF you can make a word THEN
    Add the word to the board
    Work out the points gained
    Add the points to your total
    Select more letter tiles
ENDIF
```

Of course, the IF statement will almost certainly appear within a longer set of instructions. For example, the instructions for playing our guessing game with the young child may be given as:

Note that this algorithm does not explicitly say what happens when the child makes an incorrect guess. This is because no specific action needs to be carried out when an incorrect guess is made.

```
Hide a sweet in one hand
Ask the child to guess which hand contains the sweet
Wait for the child to reply
IF the child guesses which hand the sweet is in THEN
    Give the sweet to the child
ENDIF
Ask the child if they would like to play again
```

This longer list of instructions highlights the usefulness of the term ENDIF in separating the conditional command, Give the sweet to the child, from subsequent unconditional instructions, in this case, Ask the child if they would like to play again.

Activity 1.5

A simple game involves two players. Player 1 thinks of a number between 1 and 100, then Player 2 makes a single attempt at guessing the number. Player 1 responds to a correct guess by saying *Correct*. If the guess is incorrect, Player 1 makes no response. The game is then complete and Player 1 states the value of the number.

Write the set of instructions necessary to play the game. In your solution, include the statements:

```
Player 1 says "Correct"
Player 1 thinks of a number
IF guess matches number THEN
```

The IF structure is also used in an extended form to offer a choice between two alternative actions. This expanded form of the IF statement includes another formal term, ELSE, and a second command. If the condition specified in the IF statement is true, then the command following the term THEN is executed, otherwise the

command following ELSE is carried out.

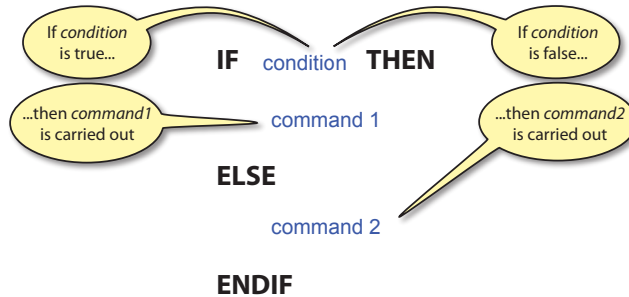
For instance, in our earlier example of playing a guessing game with a child, nothing happened if the child guessed wrongly. If the person holding the sweet were to eat it when the child's guess was incorrect, we could describe this setup with the following statement:

```
IF the child guesses which hand the sweet is in THEN
    Give the sweet to the child
ELSE
    Eat sweet yourself
ENDIF
```

The general form of this extended IF statement is shown in FIG-1.2.

FIG-1.2

The IF..THEN..ELSE
Structure



Activity 1.6

In the game of Hangman, one player has to guess the letters in a word known to the second player. At the start of the game, player 2 draws one hyphen for each letter in the word. When player 1 guesses a letter which is in the word, player two writes the letter above the appropriate hyphen. When an incorrect letter is guessed, player 2 draws a body part of a hanging man (there are 6 parts in the simple drawing).

Write an IF statement containing an ELSE section which describes the alternative actions to be taken by player 2 when player 1 guesses a letter.

In the solution include the statements:

```
Add letter at appropriate position(s)
Add part to hanged man
```

When we have several independent selections to make, then we may use several IF statements. For example, when playing Monopoly, we may buy any unpurchased property we land on. In addition, we get another turn if we throw a double. This part of the game might be described using the following statements:

```
Throw the dice
Move your piece forward by the number indicated
IF you land on an unsold property THEN
    Buy the property
ENDIF
IF you threw doubles THEN
    Throw the dice again
ELSE
    Hand the dice to the next player
ENDIF
```

Because this form of the IF statement (with or without the ELSE option) always

offers two alternative actions, the structure is known as **binary selection**.

Multi-way Selection

Although a simple IF statement can be used to select one of two alternative actions, sometimes we need to choose between more than two alternatives (known as **multi-way selection**). For example, imagine that the rules of the simple guessing game mentioned in Activity 1.5 are changed so that there are three possible responses to Player 2's guess; these being:

- Correct
- Too low
- Too high

One way to create an algorithm that describes this situation is just to employ three separate IF statements:

```
IF the guess is equal to the number you thought of THEN
    Say "Correct"
ENDIF
IF the guess is lower than the number you thought of THEN
    Say "Too low"
ENDIF
IF the guess is higher than the number you thought of THEN
    Say "Too high"
ENDIF
```

This will work, but would not be considered a good design for an algorithm since, when the first IF statement is true, we still go on and check if the conditions in the second and third IF statements are true. Checking those last two statements would be a waste of time since, if the first condition is true, the others cannot be and therefore testing them serves no purpose. Where only one of the conditions being considered can be true at a given moment in time, these conditions are known as **mutually exclusive** conditions. The most effective way to deal with mutually exclusive conditions is to check for one condition, and only if this is not true, do we bother to examine the other conditions being tested. So, for example, in our algorithm for guessing the number, we might begin by writing:

```
IF guess matches number THEN
    Say "Correct"
ELSE
    ***Check the other conditions***
ENDIF
```

Of course a statement like Check the other conditions is too vague to be much use in an algorithm (hence the asterisks to emphasise the problem). But what are these other conditions? They are the guess is lower than the number Player 1 thought of and the guess is higher than the number Player 1 thought of.

We already know how to handle a situation where there are only two alternatives: use an IF statement. So selecting between *Too low* and *Too high* requires the statement

```
IF guess is less than number THEN
    Say "Too low"
ELSE
    Say "Too high"
ENDIF
```

Now, by replacing the phrase ***Check the other conditions*** in our original algorithm with our new IF statement we get:

```
IF guess matches number THEN
    Say "Correct"
ELSE
    IF guess is less than number THEN
        Say "Too low"
    ELSE
        Say "Too high"
    ENDIF
ENDIF
```

Notice that the second IF statement is now totally contained within the ELSE section of the first IF statement. This situation is known as **nested IF statements**. Where there are even more mutually exclusive alternatives, several IF statements may be nested in this way. However, in most cases, we're not likely to need more than two nested IF statements.

Activity 1.7

In an old TV programme called *The Golden Shot*, contestants had to direct a crossbow in order to shoot an apple. The player sat at home and directed the crossbow controller via the phone. Directions were limited to the following phrases: up a bit, down a bit, left a bit, right a bit, and fire.

Write a set of nested IF statements that determine which of the above statements should be issued.

Use statements such as:

```
IF the crossbow is pointing too high THEN
and
    Say "Left a bit"
```

As you can see from the solution to Activity 1.7, although nested IF statements get the job done, the general structure can be rather difficult to follow. A better method would be to change the format of the IF statement so that several, mutually exclusive, conditions can be declared in a single IF statement along with the action required for each of these conditions. This would allow us to rewrite the solution to Activity 1.7 as:

```
IF
    crossbow is too high:      Say "Down a bit"
    crossbow is too low:      Say "Up a bit"
    crossbow is too far right: Say "Left a bit"
    crossbow is too far left: Say "Right a bit"
    crossbow is on target:    Say "Fire"
ENDIF
```

Each option is explicitly named (ending with a colon) and only the one which is true will be carried out, the others will be ignored.

Of course, we are not limited to merely five options; there can be as many as the situation requires.

When producing a program for a computer, all possibilities have to be taken into account. Early adventure games, which were text based, allowed the player to type a command such as *Go East*, *Go West*, *Go North*, *Go South* and this moved the player's character to new positions in the imaginary world of the computer program. If the

player typed in an unrecognised command such as Go North-East or Move faster, then the game would issue an error message. This setup can be described by adding an ELSE section to the structure as shown below:

```

IF
    command is Go East:
        Move player's character eastward
    command is Go West:
        Move player's character westward
    command is Go North:
        Move player's character northward
    command is Go South:
        Move player's character southward
ELSE
    Display an error message
ENDIF

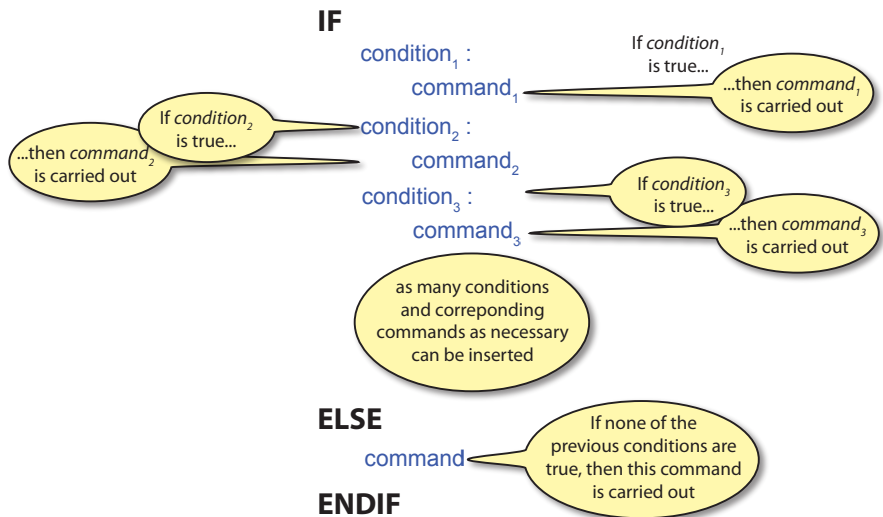
```

The additional ELSE option will be chosen only if none of the other options are applicable. In other words, it acts like a catch-all, handling all the possibilities not explicitly mentioned in the earlier conditions.

This gives us the final form of this style of the IF statement as shown in FIG-1.3.

FIG-1.3

The Multi-Way IF Structure



Activity 1.8

In the TV game Wheel of Fortune (where you have to guess a well-known phrase), you can, on your turn, either guess a consonant, buy a vowel, or make a guess at the whole phrase.

If you know the phrase, you should make a guess at what it is; if there are still many unseen letters, you should guess a consonant; as a last resort you can buy a vowel.

Write an IF statement in the style given above describing how to choose from the three options.

Complex Conditions

Often the condition given in an IF statement may be a complex one. For example, in the TV game Family Fortunes, you only win the star prize if you get 200 points and guess the most popular answers to a series of questions. This can be described in our more formal style as:

```
IF at least 200 points gained AND all most popular answers have been guessed
THEN
    winning team get the star prize
ENDIF
```

The AND Operator

Note the use of the word AND in the above example. AND (called a **Boolean operator**) is one of the terms used to link simple conditions in order to produce a more complex one (known as a **complex condition**).

The conditions on either side of the AND are called the **operands**. Both operands must be true for the overall result to be true. We can generalise this to describe the AND operator as being used in the form:

condition 1 AND condition 2

The result of the AND operator is determined using the following rules:

1. Determine the truth of condition 1
2. Determine the truth of condition 2
3. IF both conditions are true THEN
 the overall result is true
 ELSE
 the overall result is false
ENDIF

For example, if a proximity light comes on when it's dark and it detects motion then we can describe the logic of the equipment as:

```
IF it's dark AND motion has been detected THEN
    Switch on light
ENDIF
```

Now, if we assume that at a particular moment in time it's dark but no motion has been detected then the above statement would be dealt with in the manner shown in FIG-1.4.

FIG-1.4

The AND Operator

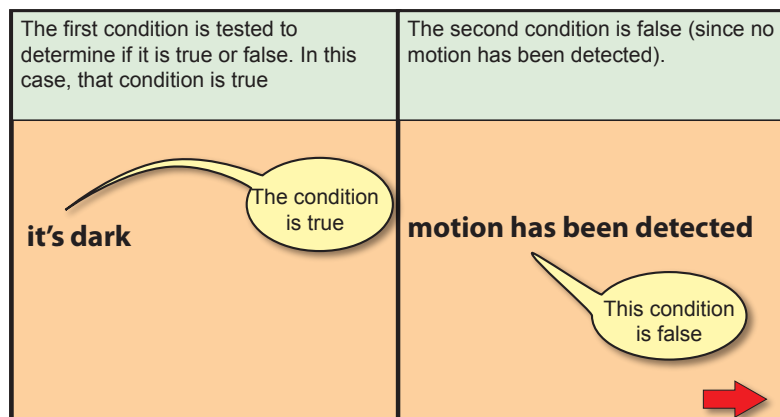


FIG-1.4
(continued)

The AND Operator

Substituting these results in the original statement we have...	Since <i>both conditions are not true</i> , we get an overall result of <i>false</i> , the command <i>Switch on light</i> is not executed.
IF <i>true</i> AND <i>false</i> THEN Switch on light ENDIF	<div>The compound condition's final value is <i>false</i> so...</div> IF <i>false</i> THEN Switch on light ENDIF <div>...command not executed</div>

With two conditions there are four possible combinations. The first possibility is that both conditions are *false*; another possibility is that condition 1 is *false* but condition 2 is *true*, etc.

Activity 1.9

What are the other possible combinations for the two conditions?

All possibilities of the AND operator are summarised in FIG-1.5.

FIG-1.5

The **AND** Truthtable

Note that the result is *true* only when both conditions are *true*.

condition 1	condition 2	condition 1 AND condition 2
false	false	false
false	true	false
true	false	false
true	true	true

Activity 1.10

In Microsoft Windows applications, the program will request the name of the file to be opened if the **Ctrl** and **O** keys are pressed together.

Write an IF statement, which includes the term **AND**, summarising this situation.

The OR Operator

Simple conditions may also be linked by the Boolean OR operator. Using OR, only one of the two conditions specified needs to be true in order to carry out the action that follows. For example, in the game of *Monopoly* you go to jail if you land on the *Go To Jail* square or if you throw three doubles in a row. This can be written as:

```
IF player landed on Go To Jail OR player has thrown 3 pairs in a row THEN
    Move player to jail
ENDIF
```

Like AND, the OR operator works on two operands:

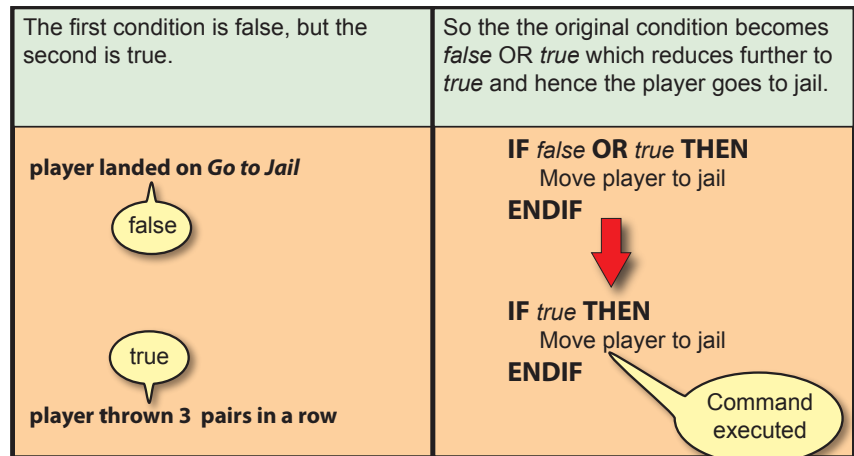
When OR is used, only one of the conditions involved needs to be true for the overall result to be true. Hence the results are determined by the following rules:

1. Determine the truth of condition 1
2. Determine the truth of condition 2
3. IF any of the conditions are true THEN
the overall result is true
ELSE
the overall result is false
ENDIF

For example, if a player in the game of *Monopoly* has not landed on the *Go To Jail* square, but has thrown three consecutive pairs, then the result of the IF statement given above would be determined as shown in FIG-1.6.

FIG-1.6

The **OR** Operator



The results of the OR operator are summarised in FIG-1.7.

FIG-1.7

The **OR** Truthtable

condition 1	condition 2	condition 1 OR condition 2
false	false	false
false	true	true
true	false	true
true	true	true

Activity 1.11

In *Monopoly*, a player can get out of jail if he throws a double or pays a £50 fine.

Express this information in an IF statement which makes use of the OR operator.

The NOT Operator

The final Boolean operator which can be used as part of a condition is NOT. This operator is used to reverse the meaning of a condition. Hence, if *it's dark* is true, then *NOT it's dark* is false. In fact, you can usually get away with just testing for the opposite condition rather than using NOT. For example, rather than write *NOT it's dark* (which isn't exactly regular English), you can write *it's light* - assuming light and dark are the only two options. Where there are many options to choose from, then

using NOT can make things a lot easier. It's a whole lot simpler to write something like

NOT day is Monday

than have to write

day is Tuesday OR day is Wednesday OR day is Thursday, etc.

Notice that the word NOT is always placed at the start of the condition and not where it would appear in everyday English (*day is NOT Monday*).

The NOT operator works on a single operand:

NOT condition

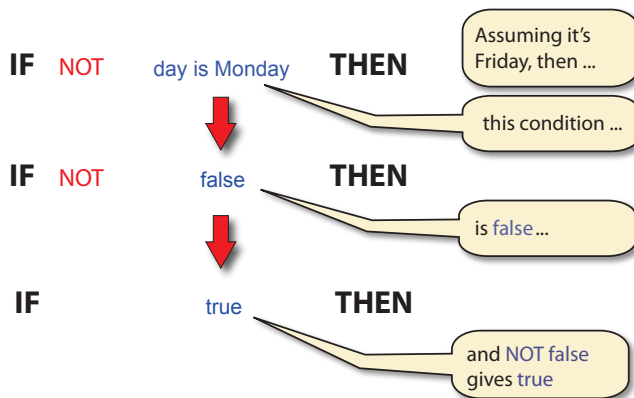
When NOT is used, the result given by the original condition (the bit without the NOT) is reversed. Hence the results are determined by the following rules:

1. Determine the truth of the original condition
2. Complement the result obtained in step 1

For example, if a player lands on a property that is not mortgaged, then the result of the IF statement given above would be determined as shown in FIG-1.8.

FIG-1.8

The NOT Operator



The results of the NOT operator are summarised in FIG-1.9.

FIG-1.9

The NOT Truthtable

condition	NOT condition
false	true
true	false

Activity 1.12

- a) Name the three types of control structures.
- b) Another term for *condition* is what?
- c) Name the two types of selection.
- d) What does the term *mutually exclusive conditions* mean?
- e) Give an example of a Boolean operator.
- f) What is a *conditional statement*?
- g) If two conditions are linked using the term AND, how many of the conditions must be true before the conditional statement is executed?

Iteration

There are certain circumstances in which it is necessary to perform the same sequence of instructions several times. For example, let's assume that a game involves throwing a dice three times and adding up the total of the values thrown. We could write instructions for such a game as follows:

```
Set the total to zero
Throw dice
Add dice value to total
Throw dice
Add dice value to total
Throw dice
Add dice value to total
Call out the value of total
```

You can see from the above that two instructions,

```
Throw dice
Add dice value to total
```

are carried out three times, once for each turn taken by the player. Not only does it seem rather time-consuming to have to write the same pair of instructions three times, but it would be even worse if the player had to throw the dice 10 times!

What is required is a way of showing that a section of the instructions is to be repeated a fixed number of times. Carrying out one or more statements over and over again is known as **looping** or **iteration**. The statement or statements we want to perform over and over again are known as the **loop body**.

Activity 1.13

What statements make up the loop body in our dice problem given above?

FOR..ENDFOR

When writing a formal algorithm in which we wish to repeat a set of statements a specific number of times, we use a FOR..ENDFOR structure. There are three separate parts to this structure. The first of these is placed just before the loop body and in it we state how often we want the statements in the loop body to be carried out. For the dice problem our statement would be:

```
FOR 3 times DO
```

Generalising, we can say this statement takes the form

```
FOR value times DO
```

where *value* would be some positive number.

Next come the statements that make up the loop body. These are indented:

```
FOR 3 times DO
  Throw dice
  Add dice value to total
```

Finally, to mark the fact that we have reached the end of the loop body statements we

add the word ENDFOR:

```
FOR 3 times DO
  Throw dice
  Add dice value to total
ENDFOR
```

Now we can rewrite our original algorithm as:

Note that ENDFOR is left-aligned with the opening FOR statement.

```
Set the total to zero
FOR 3 times DO
  Throw dice
  Add dice value to total
ENDFOR
Call out the value of total
```

The instructions between the terms FOR and ENDFOR are now carried out three times.

Activity 1.14

If the player was required to throw the dice 10 times rather than 3, what changes would we need to make to the algorithm?

If the player was required to call out the average of these 10 numbers, rather than the total, show what other changes are required to the set of instructions.

You can find the average of the 10 numbers by dividing the final total by 10.

We are free to place any statements we wish within the loop body. For example, the last version of our number guessing game produced the following algorithm:

```
Player 1 thinks of a number between 1 and 100
Player 2 makes an attempt at guessing the number
IF guess matches number THEN
  Player 1 says "Correct"
ELSE
  IF guess is less than number THEN
    Player 1 says "Too low"
  ELSE
    Player 1 says "Too high"
  ENDIF
ENDIF
```

player 2 would have more chance of winning if he were allowed several chances at guessing *player 1*'s number. To allow several attempts at guessing the number, some of the statements given above would have to be repeated.

Activity 1.15

What statements in the algorithm above need to be repeated?

To allow for 7 attempts our new algorithm becomes:

```
Player 1 thinks of a number between 1 and 100
FOR 7 times DO
  Player 2 makes an attempt at guessing the number
  IF guess matches number THEN
    Player 1 says "Correct"
  ELSE
    IF guess is less than number THEN
      Player 1 says "Too low"
    ELSE
      Player 1 says "Too high"
    ENDIF
  ENDIF
ENDFOR
```

```
ENDIF
ENDIF
ENDFOR
```

Activity 1.16

Can you see a flaw in the algorithm?

If not, try playing the game a few times, playing exactly according to the instructions in the algorithm.

Activity 1.17

During a lottery draw, two actions are performed exactly 6 times. These are:

- Pick out ball
- Call out number on the ball

Add a FOR loop to the above statements to create an algorithm for the lottery draw process.

Occasionally, we may have to use a slightly different version of the FOR loop. Imagine we are trying to write an algorithm explaining how to decide who goes first in a game. In this game every player throws a dice and the player who throws the highest value goes first. To describe this activity we know that each player does the following task:

Player throws dice

But since we can't know in advance how many players there will be, we write the algorithm using the statement

FOR every player DO

to give the following algorithm

```
FOR every player DO
  Throw dice
ENDFOR
Player with highest throw goes first
```

If we had to save the details of a game of chess with the intention of going back to the game later, we might write:

```
FOR each piece on the board DO
  Write down the name and position of the piece
ENDFOR
```

Activity 1.18

A game uses cards with images of warriors. At one point in the game the player has to remove from his hand every card with an image of a knight. To do this the player must look through every card and, if it is a knight, remove the card.

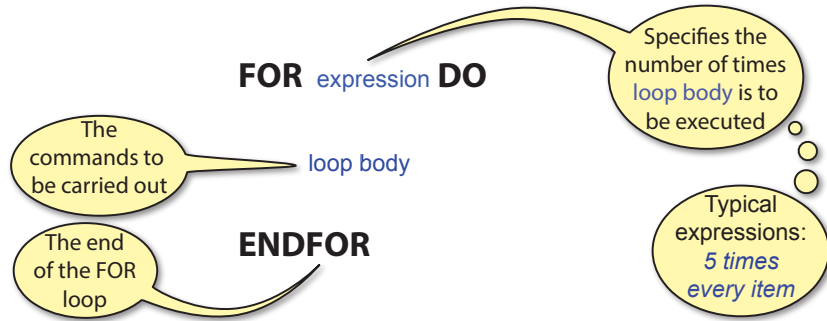
Write down a set of instructions which performs the task described above. Your solution should include the statements

```
FOR every card in player's hand DO    and    IF card is a knight THEN
```


The general form of the FOR statement is shown in FIG-1.10.

FIG-1.10

The FOR..ENDFOR
Loop



Although the FOR loop allows us to perform a set of statements a specific number of times, this statement is not always suitable for the problem we are trying to solve.

For example, in the guessing game of Activity 1.15 we stated that the loop body was to be performed 7 times, but what if player 2 guesses the number after only three attempts? If we were to follow the algorithm exactly (as a computer would), then we must make four more guesses at the number even after we know the correct answer!

To solve this problem, we need another way of expressing looping which does not commit us to a specific number of iterations.

REPEAT.. UNTIL

The REPEAT .. UNTIL statement allows us to specify that a set of statements should be repeated until some condition becomes true, at which point iteration should cease.

The word REPEAT is placed at the start of the loop body and, at its end, we add the UNTIL statement. The UNTIL statement also contains a condition, which, when true, causes iteration to stop. This is known as the **terminating** (or exit) **condition**. For example, we could use the REPEAT.. UNTIL structure rather than the FOR loop in our guessing game algorithm. The new version would then be:

```
Player 1 thinks of a number between 1 and 100
REPEAT
  Player 2 makes an attempt at guessing the number
  IF guess matches number THEN
    Player 1 says "Correct"
  ELSE
    IF guess is less than number THEN
      Player 1 says "Too low"
    ELSE
      Player 1 says "Too high"
    ENDIF
  ENDIF
UNTIL player 2 guesses correctly
```

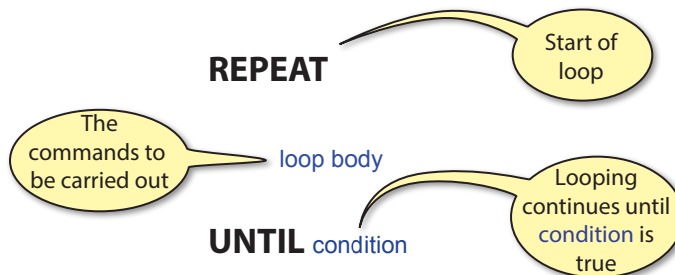
We could also use the REPEAT..UNTIL loop to describe how a slot machine (one-armed bandit) is played:

```
REPEAT
  Put coin in machine
  Pull handle
  IF you win THEN
    Collect winnings
  ENDIF
UNTIL you want to stop
```

The general form of this structure is shown in FIG-1.11.

FIG-1.11

The REPEAT..UNTIL
Loop



The terminating condition may use the Boolean operators AND, OR and NOT as well as parentheses, where necessary.

Activity 1.19

Confronted with a pile of unordered books when looking for a specific publication, the only way to find the desired title is to examine each book in turn until the required one is found. Of course, there's a possibility that the book is not in the pile.

Using REPEAT..UNTIL, write the logic required to search for the book.

Returning to the number guessing game on the previous page, there is still a problem. By using a REPEAT .. UNTIL loop we are allowing *player 2* to have as many guesses as needed to determine the correct number. That doesn't lead to a very interesting game. Later we'll discover how we might solve this problem.

WHILE.. ENDWHILE

A final method of iteration, differing only subtly from the REPEAT.. UNTIL loop, is the WHILE .. ENDWHILE structure which has an **entry condition** at the start of the loop. The following example illustrates the usefulness of this new structure.

The aim of the card game of Pontoon is to attempt to make the value of your cards add up to 21 without going over that value. Each player is dealt two cards initially but can repeatedly ask for more cards by saying "twist". One player is designated the dealer. The dealer must twist while his cards have a total value of less than 16. So we might write the rules for the dealer as:

```
Calculate the sum of the initial two cards
REPEAT
    Take another card
    Add new card's value to sum
UNTIL sum is greater than or equal to 16
```

But there's a problem with the solution: if the sum of the first two cards is already 16 or above, we still need to take a third card (just work through the logic, if you can't see why). By using the WHILE..ENDWHILE structure we could describe the logic as

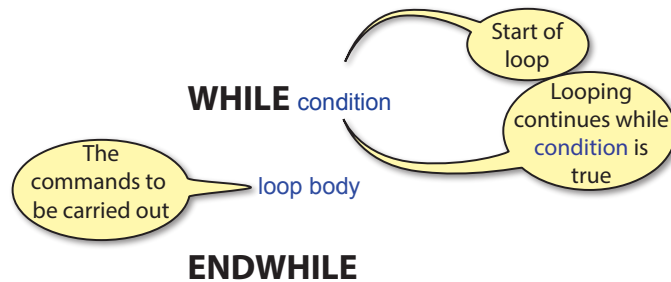
```
Calculate sum of the initial two cards
WHILE sum is less than 16 DO
    Take another card
    Add new card's value to sum
ENDWHILE
```

Now determining if the sum is less than 16 is performed before the *Take another card* instruction. If the dealer's two cards already add up to 16 or more, then the *Take another card* instruction will be ignored.

The general form of the WHILE.. ENDWHILE statement is shown in FIG-1.12.

FIG-1.12

The WHILE..
ENDWHILE Loop



In what way does this differ from the REPEAT statement? There are two differences:

- The condition is given at the beginning of the loop.
- Looping stops when the condition is false.

The main consequence of this is that it is possible to bypass the loop body of a WHILE structure entirely without ever carrying out any of the instructions it contains.

On the other hand, the loop body of a REPEAT structure will always be executed at least once.

Activity 1.20

A game involves throwing two dice. If the two values thrown are not the same, then the dice showing the lower value must be rolled again. This process is continued until both dice show the same value.

Write a set of instructions to perform this game.

Your solution should contain the statements

and Roll both dice
 Choose dice with lower value

Activity 1.21

- a) What is the meaning of the term **iteration**?
- b) Name the three types of looping structures.
- c) What type of loop structure should be used when looping needs to occur an exact number of times?
- d) What type of loop structure can bypass its loop body without ever executing it?
- e) What type of loop contains an exit condition?

Infinite Loops

If a loop can never exit, it is known as an **infinite loop**. As a general rule, infinite loops are caused by some error in the logic. For example, the algorithm

```
Think of a number
REPEAT
    Subtract 1 from the number
UNTIL the number is zero
```

will never be completed if the number you start with is already zero or less.

Data

We know we need to retain information. Look at your phone; packed with names, email addresses, phone numbers, and much more. Even when playing an old-fashioned board game we need to remember things such as the number you threw on the dice, where your piece is on the board and so on. These examples introduce the need to process facts and figures (known as **data**).

Every item of data has two basic characteristics :

and a name
 a value

The name of a data item is a description of the type of information it represents. Hence on a form we might see boxes labelled as *Forename*, *Surname*, *Address*, *Phone No*, etc. These are the data names. And, when we've completed the form, the boxes will contain the values we have entered. These entries are the data values. In programming, a data item is often referred to as a **variable**. This term arises from the fact that, although the name assigned to a data item cannot change, its value may vary. For example, the value assigned to a variable called *salary* may rise (or fall) over weeks, months or years.

Types of Data

Most computer programming languages need to be told what type of value is to be held in a variable - for example, it needs to know if a variable will hold a number or a message. Once the variable is set up for one type of value, it can't be used to hold any other type. Three of the basic data types recognised by a language such as AGK BASIC are:

integer	holds whole numbers only (eg -12, 0, 92).
real	(also known as floating point numbers) holds numbers containing fractions (-14.6, 0.005, 176.0) - notice that the fraction part may be .0.
string	holds zero or more characters. A character may be alphabetic, numeric, or punctuation marks (A, 7, *).

Other data types are possible, but we'll look at these in a later chapter.

Operations on Data

There are four basic operations that a computer can do with data. These are:

Input

This involves being given a value for a data item. For example, in our number-guessing game, the player who has thought of the original number is given the value

of the guess from the second player. When playing Noughts and Crosses adding an X (or O) changes the set up on the board. When using a computer, any value entered at the keyboard, or any movement or action dictated by a mouse or joystick would be considered as data entry. This type of action is known as an **input operation**.

Calculation

Most games involve some basic arithmetic. In Monopoly, the banker has to work out how much change to give a player buying a property. If a character in an adventure game is hit, points must be deducted from his strength value. This type of instruction is referred to as a **calculation operation**.

Comparison

Often values have to be compared. For example, we need to compare the two numbers in our guessing game to find out if they are the same. This is known as a **comparison operation**.

Output

The final requirement is to communicate with others to give the result of some calculation or comparison. For example, in the guessing game, player 1 communicates with player 2 by saying either that the guess is *Correct*, *Too high* or *Too low*.

In a computer environment, the equivalent operation would normally involve displaying information on a screen or printing it on paper. For instance, in a racing game your speed and time will be displayed on the screen. This is called an **output operation**.

When describing a calculation, it is common to use arithmetic operator symbols rather than English. Hence, instead of writing the word subtract we use the minus sign (-). However, programming languages use a slightly different set of symbols than standard mathematics (see FIG-1.13).

FIG-1.13

The Arithmetic Operators

English	Symbol
Multiply	*
Divide	/
Add	+
Subtract	-

Similarly, when we need to compare values, rather than use terms such as *is less than*, we use the *less than* symbol (<). A summary of these relational operators is given in FIG-1.14.

FIG-1.14

The Relational Operators

English	Symbol
is less than	<
is less than or equal to	<=
is greater than	>
is greater than or equal to	>=
is equal to	=
is not equal to	<>

As well as replacing the words used for arithmetic calculations and comparisons with

symbols, the term *calculate* or *set* is often replaced by the shorter but more cryptic symbol \rightarrow between the variable being assigned a value and the value itself. Using this abbreviated form, the instruction:

Calculate time to complete course as distance divided by speed

becomes

`time \rightarrow distance / speed`

Although the long-winded English form is more readable, this more cryptic style is briefer and is much closer to the code used when programming a computer.

Below we compare the two methods of describing our guessing game; first in English:

```
Player 1 thinks of a number between 1 and 100
REPEAT
  Player 2 makes an attempt at guessing the number
  IF guess matches number THEN
    Player 1 says "Correct"
  ELSE
    IF guess is less than number THEN
      Player 1 says "Too low"

      ELSE
        Player 1 says "Too high"
    ENDIF
  ENDIF
UNTIL player 2 guesses correctly
```

Using some of the symbols described earlier, we can rewrite this as:

```
Player 1 thinks of a number between 1 and 100
REPEAT
  Player 2 makes an attempt at guessing the number
  IF guess = number THEN
    Player 1 says "Correct"
  ELSE
    IF guess < number THEN
      Player 1 says "Too low"
    ELSE
      Player 1 says "Too high"
    ENDIF
  ENDIF
UNTIL guess = number
```

Activity 1.22

- a) What are the two main characteristics of any data item?
- b) When data is input, from where is its value obtained?
- c) Give an example of a relational operator.

Levels of Detail

When we start to write an algorithm in English, one of the things we need to consider is exactly how much detail should be included. For example, we might describe how to record a video on a digital camcorder as:

```
Insert memory stick
Choose appropriate recording settings
```

However, this lacks enough detail for anyone unfamiliar with the operation of the machine. Therefore, we could replace the first statement with:

```
Open the flap covering the memory chip slot
IF there is a chip already in the slot THEN
    Remove it
ENDIF
Place the new memory stick in slot
Close flap
```

and the second statement could be substituted by:

```
Set recording quality
Set exposure to automatic
Set focus to automatic
```

This approach of starting with a less detailed sequence of instructions and then, where necessary, replacing each of these with more detailed instructions can be used to good effect when tackling long and complex problems. By using this technique, we are defining the original problem as an equivalent sequence of simpler problems before going on to create a set of instructions to handle each of these simpler problems. This divide-and-conquer strategy is known as **stepwise refinement**. The following is a fully worked example of this technique:

Problem:

Describe how to make a cup of tea.

Outline Solution:

1. Fill kettle
2. Boil water
3. Put tea bag in teapot
4. Add boiling water to teapot
5. Wait 1 minute
6. Pour tea into cup
7. Add milk and sugar to taste

This is termed a **LEVEL 1 solution**.

As a guideline we should aim for a LEVEL 1 solution with between 5 and 12 instructions. Notice that each instruction has been numbered. This is merely to help with identification during the stepwise refinement process.

Before going any further, we must assure ourselves that this is a correct and full (though not detailed) description of all the steps required to tackle the original problem. If we are not happy with the solution, then changes must be made before going any further.

Next, we examine each statement in turn and determine if it should be described in more detail. Where this is necessary, rewrite the statement to be dealt with, and below it, give the more detailed version. For example. Fill kettle would be expanded thus:

1. Fill kettle
 - 1.1 Remove kettle lid
 - 1.2 Put kettle under tap
 - 1.3 Turn on tap
 - 1.4 When kettle is full, turn off tap
 - 1.5 Replace lid on kettle

The numbering of the new statement reflects that they are the detailed instructions

pertaining to statement 1. Also note that the number system is not a decimal fraction, so if there were to be many more statements they would be numbered 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, etc.

It is important that these sets of more detailed instructions describe how to perform only the original task being examined - they must achieve no more and no less. Sometimes the detailed instructions will contain control structures such as IFs, WHILEs or FORs. Where this is the case, the whole structure must be included in the detailed instructions for that task. Having satisfied ourselves that the breakdown is correct, we proceed to the next statement from the original solution.

- 2. Boil water
 - 2.1 Plug in kettle
 - 2.2 Switch on power at socket
 - 2.3 Switch on power at kettle
 - 2.4 When water boils switch off kettle

The next two statements expand as follows:

- 3. Put tea bag in teapot
 - 3.1 Remove lid from teapot
 - 3.2 Add tea bag to teapot
- 4. Add boiling water to teapot
 - 4.1 Take kettle over to teapot
 - 4.2 Add required quantity of water from kettle to teapot

But not every statement from a level 1 solution needs to be expanded. In our case there is no more detail to add to the statement

- 5. Wait 1 minute

and therefore, we leave it unchanged.

The last two statements expand as follows:

- 6. Pour tea into cup
 - 6.1 Take teapot over to cup
 - 6.2 Pour required quantity of tea from teapot into cup
- 7. Add milk and sugar as required
 - 7.1 IF milk is required THEN
 - 7.2 Add milk
 - 7.3 ENDIF
 - 7.4 IF sugar is required THEN
 - 7.5 Add sugar
 - 7.6 Stir tea
 - 7.7 ENDIF

Notice that this last expansion (step 7) has introduced IF statements. Control structures (i.e. IF, WHILE, FOR, etc.) can be introduced at any point in an algorithm.

Finally, we can describe the solution to the original problem in more detail by substituting the statements in our LEVEL 1 solution by their more detailed equivalent:

- 1.1 Remove kettle lid
- 1.2 Put kettle under tap
- 1.3 Turn on tap
- 1.4 When kettle is full, turn off tap
- 1.5 Place lid back on kettle
- 2.1 Plug in kettle
- 2.2 Switch on power at socket
- 2.3 Switch on power at kettle


```

2.4 When water boils switch off kettle
3.1 Remove lid from teapot
3.2 Add tea bag to teapot
4.1 Take kettle over to teapot
4.2 Add required quantity of water from kettle to teapot
5. Wait 1 minute
6.1 Take teapot over to cup
6.2 Pour required quantity of tea from teapot into cup
7.1 IF milk is required THEN
7.2     Add milk
7.3 ENDIF
7.4 IF sugar is required THEN
7.5     Add sugar
7.6     Stir tea
7.7 ENDIF

```

This is a LEVEL 2 solution. Note that a level 2 solution includes any LEVEL 1 statements which were not given more detail (in this case, *Wait 1 minute*).

For some more complex problems it may be necessary to repeat this process to more levels before sufficient detail is achieved. That is, statements in LEVEL 2 may be given more detail in a LEVEL 3 breakdown.

Activity 1.23

The game of battleships involves two players. Each player draws two 10 by 10 grids. Each of these have columns lettered A to J and rows numbered 1 to 10. In the first grid each player marks the position of warships. Ships are added as follows:

1 aircraft carrier	4 squares
2 destroyers	3 squares each
3 cruisers	2 squares each
4 submarines	1 square each

The squares of each ship must be adjacent and must be vertical or horizontal. The first player now calls out a grid reference.

The second player responds to the call by saying HIT or MISS. HIT is called if the grid reference corresponds to a position of a ship. The first player then marks this result on his second grid using an o to signify a miss and x for a hit (see diagram below).

	A	B	C	D	E	F	G	H	I	J
1										
2										
3				A	A	A	A			
4									S	
5	C	C						D		
6				S				D		
7		D	D	D				D		
8						C			S	
9		S				C				
10				C	C					

	A	B	C	D	E	F	G	H	I	J
1										O
2										
3							O			
4										
5										
6			X	X	X					
7								O		
8										
9										
10										

Vessels are positioned
in the left-hand grid

Results of guesses are
placed in the right-hand grid

continued on next page

Activity 1.23 (continued)

If the first player achieves a HIT then he continues to call grid references until MISS is called. In response to a HIT or MISS call the first player marks the second grid at the reference called: 0 for a MISS, X for a HIT.

When the second player responds with MISS the first player's turn is over, and the second player has his turn.

The first player to eliminate all segments of the opponent's ships is the winner. However, each player must have an equal number of turns, and if both sets of ships are eliminated in the same round the game is a draw.

The algorithm describing the task of one player is given in the instructions below. Create a LEVEL 1 algorithm by assembling the lines in the correct order, adding line numbers to the finished description.

```
Add ships to left grid
UNTIL there is a winner
Call grid position(s)
REPEAT
Respond to other player's call(s)
Draw grids
```

To create a LEVEL 2 algorithm, some of the above lines will have to be expanded to give more detail. More detailed instructions are given below for the statements Call grid position(s) and Respond to other player's call(s).

By reordering and numbering the lines below create LEVEL 2 details for these two statements.

```
UNTIL other player misses
Mark position in second grid with X
Get other player's call
Get reply
Get reply
ENDIF
Call HIT
Call MISS
Mark position in second grid with 0
WHILE reply is HIT DO
Call grid reference
Call grid reference
IF other player's call matches position of ship THEN
ENDWHILE
REPEAT
ELSE
```

Checking for Errors

Once we've created our algorithm we would like to make sure it is correct. Unfortunately, there is no foolproof way to do this! But we can at least try to find any errors or omissions in the set of instructions we have created.

We do this by going back to the original description of the task our algorithm is attempting to solve. For example, let's assume we want to check our number guessing

game algorithm. In the last version of the game we allowed the second player to make as many guesses as required until he came up with the correct answer. The first player responded to each guess by saying either “Too low”, “Too high” or “Correct”.

To check our algorithm for errors we must come up with typical values that might be used when carrying out the set of instructions and those values should be chosen so that each possible result is achieved at least once.

So, as well as making up values, we need to predict what response our algorithm should give to each value used. Hence, if the first player thinks of the value 42 and the second player guesses 75, then the first player will respond to the guess by saying “Too high”.

Our set of test values must evoke each of the possible results from our algorithm. One possible set of values and the responses are shown in FIG-1.15.

FIG-1.15

Test Data for the
Number Guessing Game
Algorithm

Test Data	Expected Results
number = 42	
guess = 75	Says “Too high”
guess = 15	Says “Too low”
guess = 42	Says “Correct”

Once we’ve created test data, we need to work our way through the algorithm using that test data and checking that we get the expected results. The algorithm for the number game is shown below, this time with instruction numbers added.

1. Player 1 thinks of a number between 1 and 100
2. REPEAT
3. Player 2 makes an attempt at guessing the number
4. IF guess = number THEN
5. Player 1 says “Correct”
6. ELSE
7. IF guess < number THEN
8. Player 1 says “Too low”
9. ELSE
10. Player 1 says “Too high”
11. ENDIF
12. ENDIF
13. UNTIL guess = number

Next we create a new table (called a **trace table**) with the headings as shown in FIG-1.16.

FIG-1.16

A Trace Table

Instruction	Condition	T/F	Variables <i>number guess</i>	Output

Now we work our way through the statements in the algorithm filling in a line of the trace table for each instruction.

Instruction 1 is for player 1 to think of a number. Using our test data, that number will be 42, so our trace table starts with the line shown in FIG-1.17.

FIG-1.17

Working through a
Trace 1

Instruction	Condition	T/F	Variables number guess	Output
1			42	

The REPEAT word comes next. Although this does not cause any changes, nevertheless a 2 should be entered in the next line of our trace table. Instruction 3 involves player 2 making a guess at the number (this guess will be 75 according to our test data). After 3 instructions our trace table is as shown in FIG-1.18.

FIG-1.18

Working through a
Trace 2

Instruction	Condition	T/F	Variables number guess	Output
1			42	
2				
3			75	

Instruction 4 is an IF statement containing a condition. This condition and its result are written into columns 2 and 3 as shown in FIG-1.19.

FIG-1.19

Working through a
Trace 3

Instruction	Condition	T/F	Variables number guess	Output
1			42	
2				
3			75	
4	guess = number	F		

Because the condition is false, we now jump to instruction 6 (the ELSE line) and on to 7. This is another IF statement and our table now becomes that shown in FIG-1.20.

FIG-1.20

Working through a
Trace 4

Instruction	Condition	T/F	Variables number guess	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		

Since this second IF statement is also false, we move on to statements 9 and 10. Instruction 10 causes output (speech) and hence we enter this in the final column as shown in FIG-1.21.

FIG-1.21

Working through a
Trace 5

Instruction	Condition	T/F	Variables number guess	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		
9				
10				Too high

Now we move on to statements 11,12 and 13 as shown in FIG-1.22.

FIG-1.22

Working through a
Trace 6

Instruction	Condition	T/F	Variables number guess	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		
9				
10				Too high
11				
12				
13	guess = number	F		

Since statement 13 contains a condition which is false, we return to statement 2, executing it and then moving on to 3 where we enter 15 as our second guess (see FIG-1.23).

FIG-1.23

Working through a
Trace 7

Instruction	Condition	T/F	Variables number guess	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		
9				
10				Too high
11				
12				
13	guess = number	F		
2				
3			15	

This method of checking is known as **desk checking** or **dry running**.

Activity 1.24

Create your own trace table for the number-guessing game and, using the same test data as given in FIG-1.15 complete the testing of the algorithm.

Were the expected results obtained?

Summary

- Computers can perform many tasks by executing different programs.
- An algorithm is a sequence of instructions which solves a specific problem.
- A program is a sequence of computer instructions which usually manipulates data and produces results.
- Three control structures are used in programs :
 - Sequence
 - Selection
 - Iteration

- A sequence is a list of instructions which are performed one after the other.
- Selection involves choosing between two or more alternative actions.
- Selection is performed using the IF statement.
- There are three forms of IF statement:

```
IF condition THEN
    instructions
ENDIF
```

```
IF condition THEN
    instructions
ELSE
    instructions
ENDIF
```

```
IF
    condition 1:
        instructions
    condition 2:
        instructions
    condition x :
        instructions
ELSE
    instructions
ENDIF
```

- Iteration is the repeated execution of one or more statements.
- Iteration is performed using one of three instructions:

```
FOR number of iterations required DO
    instructions
ENDFOR
REPEAT
    instructions
UNTIL condition
```

```
WHILE condition DO
    instructions
ENDWHILE
```

- A condition is an expression which is either *true* or *false*.
- Simple conditions can be linked using AND or OR to produce a complex condition.
- The meaning of a condition can be reversed by adding the word NOT.
- Data items (or variables) hold the information used by the algorithm.
Data item values may be:

```
Input
Calculated
Compared
or      Output
```

- Calculations can be performed using the following arithmetic operators:

Multiplication	*
Addition	+
Division	/
Subtraction	-

- The order of priority of an operator may be overridden using parentheses.
- Comparisons can be performed using the relational operators:

Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	=
Not equal to	<>

- The symbol -> is used to assign a value to a data item. Read this symbol as is assigned the value.
- In programming, a data item is referred to as a variable.
- The divide-and-conquer strategy of stepwise refinement can be used when creating an algorithm.
- LEVEL 1 solution gives an overview of the sub-tasks involved in carrying out the required operation.
- LEVEL 2 gives a more detailed solution by taking each sub-task from LEVEL 1 and, where necessary, giving a more detailed list of instructions required to perform that sub-task.
- Not every statement needs to be broken down into more detail.
- Further levels of detail may be necessary when using stepwise refinement for complex problems.
- Further refinement may not be required for every statement.
- An algorithm can be checked for errors or omissions using a trace table.

Solutions

Activity 1.1

No solution required.

Activity 1.2

One possible solution is:

```
Fill A
Fill B from A
Empty B
Empty A into B
Fill A
Fill B from A
```

Activity 1.3

- a) An algorithm
- b) A computer program
- c) mips (millions of instructions per second)

Activity 1.4

```
Choose club
Take up correct stance beside ball
Grip club correctly
Swing club backwards
Swing club forwards, attempting to hit ball
```

The second and third statements could be interchanged.

Activity 1.5

```
Player 1 thinks of a number
Player 2 makes a guess at the number
IF guess matches number THEN
    Player 1 says "Correct"
ENDIF
Player 1 states the value of the number
```

Activity 1.6

```
IF letter appears in word THEN
    Add letter at appropriate position(s)
ELSE
    Add part to hanged man
ENDIF
```

Activity 1.7

```
IF the crossbow is on target THEN
    Say "Fire"
ELSE
    IF the crossbow is pointing too high THEN
        Say "Down a bit"
    ELSE
        IF the crossbow is pointing too low THEN
            Say "Up a bit"
        ELSE
            IF the crossbow is too far left THEN
                Say "Right a bit"
            ELSE
                Say "Left a bit"
            ENDIF
        ENDIF
    ENDIF
ENDIF
```

Activity 1.8

```
IF
    you know the phrase:
        Make guess at phrase
        there are many unseen letters:
            Guess a consonant
ELSE
    Buy a vowel
ENDIF
```

Activity 1.9

Other possibilities are:

Both conditions are true
condition 1 is true and condition 2 is false

Activity 1.10

```
IF Ctrl key pressed AND O key pressed THEN
    Request filename
ENDIF
```

Activity 1.11

```
IF double thrown OR fine paid THEN
    Player gets out of jail
ENDIF
```

Activity 1.12

- a) Sequence
Selection
Iteration
- b) Boolean expression
- c) Binary selection Multi-way selection
- d) No more than one of the conditions can be true at any given time.
- e) Boolean operators are: AND, OR, and NOT.
- f) A conditional statement is a statement which is executed only if a given set of conditions are met.
- g) Both conditions must be true.

Activity 1.13

```
Throw dice
Add dice value to total
```

Activity 1.14

Only one line, the FOR statement, would need to be changed, the new version being:

```
FOR 10 times DO
```

To call out the average, the algorithm would change to

```
Set the total to zero
FOR 10 times DO
    Throw dice
    Add dice value to total
ENDFOR
Calculate average as total divided by 10
Call out the value of average
```

Activity 1.15

In fact, only the first line of our algorithm is not repeated, so the lines that need to be repeated are:

```
Player 2 makes an attempt at guessing the number
IF guess matches number THEN
    Player 1 says "Correct"
ELSE
    IF guess is less than number THEN
        Player 1 says "Too low"
    ELSE
        Player 1 says "Too high"
    ENDIF
ENDIF
```

Activity 1.16

The FOR loop forces the loop body to be executed exactly 7 times. If the player guesses the number in less attempts, the algorithm will nevertheless continue to ask for the remainder of the 7 guesses.

Later, we'll see how to solve this problem.

Activity 1.17

```
FOR 6 times DO
  Pick out ball
  Call out number on the ball
ENDFOR
```

Activity 1.18

```
FOR every card in player's hand DO
  IF card is a knight THEN
    Remove card from hand
  ENDIF
ENDFOR
```

Activity 1.19

```
REPEAT
  Read next book title
UNTIL required title found OR no books remaining
```

Activity 1.20

```
Roll both dice
WHILE dice values don't match DO
  Choose dice with lower value
  Throw chosen dice
ENDWHILE
```

Note that the WHILE line could have been written as

```
WHILE NOT dice values match DO
```

Activity 1.21

- Iteration** means executing a set of statements repeatedly.
- FOR..ENDFOR, REPEAT..UNTIL and WHILE..ENDWHILE
- The FOR..ENDFOR structure.
- The WHILE..ENDWHILE structure.
- The REPEAT..UNTIL structure.

Activity 1.22

- Its name and value.
- From outside the system. In a computerised setup, this is often entered from a keyboard.
- The relational operators are:
 - < (less than)
 - <= (less than or equal to)
 - > (greater than)
 - >= (greater than or equal to)
 - = (equal to)
 - <> (not equal to)

Activity 1.23

The LEVEL 1 is coded as:

- Draw grids
- Add ships to left grid
- REPEAT
- Call grid position(s)
- Respond to other player's call(s)
- UNTIL there is a winner

The expansion of statement 4 would become:

- Call grid reference
- Get reply
- WHILE reply is HIT DO
- Mark position in second grid with X
- Call grid reference
- Get reply

4.7 ENDWHILE

4.8 Mark position in second grid with 0

The expansion of statement 5 would become:

5.1 REPEAT

5.2 Get other player's call

5.3 IF other player's call matches position of ship THEN

5.4 Call HIT

5.5 ELSE

5.6 Call MISS

5.7 ENDIF

5.8 UNTIL other player misses

Activity 1.24

Instruction	Condition	T/F	Variables number guess	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		
9				
10				Too high
11				
12				
13	guess = number	F		
2				
3			15	
4	guess = number	F		
6				
7	guess < number	T		
8				Too low
11				
12				
13	guess = number	F		
2				
3			42	
4	guess = number	T		
5				
12				
13	guess = number	T		Correct

2

Starting AGK

In this Chapter:

- ☐ Understanding Compilation
- ☐ Getting Started with AGK
- ☐ Creating a First Project
- ☐ Installing an App on a Device
- ☐ Creating Output
- ☐ Adding Comments
- ☐ Changing Output Colour, Size and Spacing
- ☐ Adjust an App Window's Properties
- ☐ Adding a Splash Screen

Programming a Computer

Introduction

In the last chapter we created algorithms written in a style known as **structured English**. But if we want to create an algorithm that can be followed by a computer, then we need to convert our structured English instructions into a programming language.

A housekeeping program is one which performs mundane chores such as file copying, data communications, etc. and has little user input.

There are many programming languages; C, C++, Java, C#, and Visual Basic being amongst the most widely used. So how do we choose which programming language to use? Each language has its own strengths. For example, Java allows multi-platform programs to be created easily, while C is ideal for creating housekeeping applications. So, when we choose a programming language, we want one that is best suited to the task we have in mind.

We are going to use a programming language known as AGK BASIC. This language was designed specifically for writing computer games which can then be used on a wide range of devices - anything from your regular computer to a tablet or even a smartphone. Because of this, AGK BASIC has many unique commands for displaying graphics on various screen resolutions and for handling a wide range of input methods - anything from a standard mouse to a touch screen or an accelerometer.

The Compilation Process

When you begin the process of creating a game using AGK, several files are automatically created. One of these files is designed to hold your program code; the others hold additional details required by the project. These extra files have their contents created automatically by AGK so we need not worry about them at this stage.

Because each game that we create consists of several files, we refer to this collection of files as a **project**. One of these files (always named *main.agc* in every project) contains the actual program code.

Each new project is automatically assigned its own folder.

As we will soon see, the programming language AGK BASIC uses statements that retain some English terms and phrases. This means we can look at the set of instructions and make some sense of what is happening after only a relatively small amount of training.

Unfortunately, the processor inside a digital device (computer, tablet, or smartphone) understands only instructions given in the form of a sequence of 1's and 0's in a format known as **machine code**. The device has no capability of directly following a set of instructions written in AGK BASIC. But this need not be a problem; we simply need to translate the AGK BASIC statements into machine code (just as we might have a piece of text translated from Russian to English).

We begin the process of creating a new piece of software by mentally converting our structured English algorithm (which we will have already created) into a sequence of AGK BASIC statements. These statements are entered using a text editor which is nothing more than a simple word-processor-like program allowing such basic operations as inserting and deleting text. Once the complete program has been entered, we get the machine itself to translate those instructions from AGK BASIC

into **machine code**. The original program code is known as the **source code**; the machine code is known as the **object code** and the saved version of this as the **executable file**.

The translator (known as a **compiler**) is simply another program installed in the computer. After typing in our program instructions, we feed these to the compiler which produces the equivalent instructions in machine code. These instructions are then executed by the computer and we should see the results of our calculations appear on the screen (assuming there are output statements in the program).

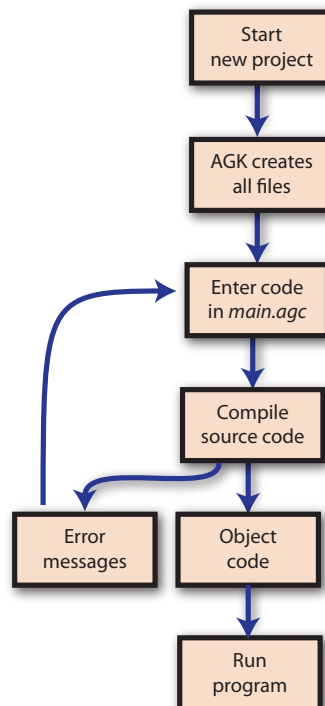
The compiler is a very exacting task master. The structure, or **syntax**, of every statement must be exactly right. If you make the slightest mistake, even something as simple as missing out a comma or misspelling a word, the translation process will fail. When this happens in AGK, a window appears giving details of the error. A failure of this type is known as a **syntax error** - a mistake in the grammar of your commands. Any syntax errors have to be corrected before you can try compiling the program again.

When you are working on a project, it is best to save your work at regular intervals. That way, if there is a power cut, you won't have lost all your code!

When the program code is complete, we compile our program (translating it from source code to object code). When the translation process is finished, yet another file is produced. This new file (which has an *.exe* extension), contains the object code. To run our program, the source code in the executable file is loaded into the computer's memory (RAM) and the instructions it contains are carried out. The whole process is summarised in FIG-2.1.

FIG-2.1

The Compilation
Process



If we want to make changes to the program, we load the source code into the editor, make the necessary modifications, then save and recompile our program, thereby replacing the old version of source and executable files.

Activity 2.1

- a) What type of instructions are understood by a computer?
- b) What piece of software is used to translate a program from source code to object code?
- c) Misspelling a word in your program is an example of what type of error?

Starting AGK

Introduction

AGK is an Integrated Development Environment (IDE) software package designed to create 2D games that can then be run on various hardware devices. IDE simply means that the editing, compiling and testing are all achieved while working from within a single package.

AGK allows programs to be written in either BASIC or C++. This book covers only the BASIC language aspect of AGK.

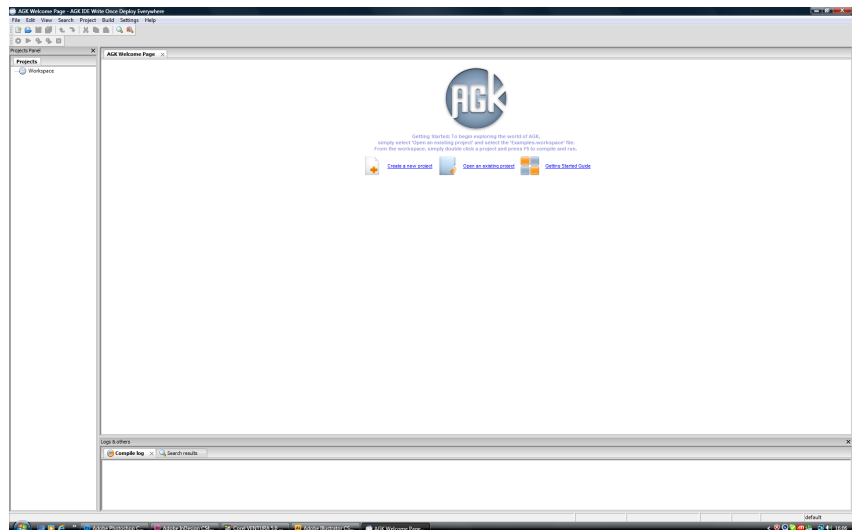
AGK was created by Lee Bamber, CEO of The Game Creators Ltd and was derived from his earlier creation, DarkBASIC which is a programming language designed to develop games for the PC platform only.

Starting Up AGK

Once you've installed AGK, running the package will present you with the screen shown in FIG-2.2.

FIG-2.2

The AGK Startup Screen

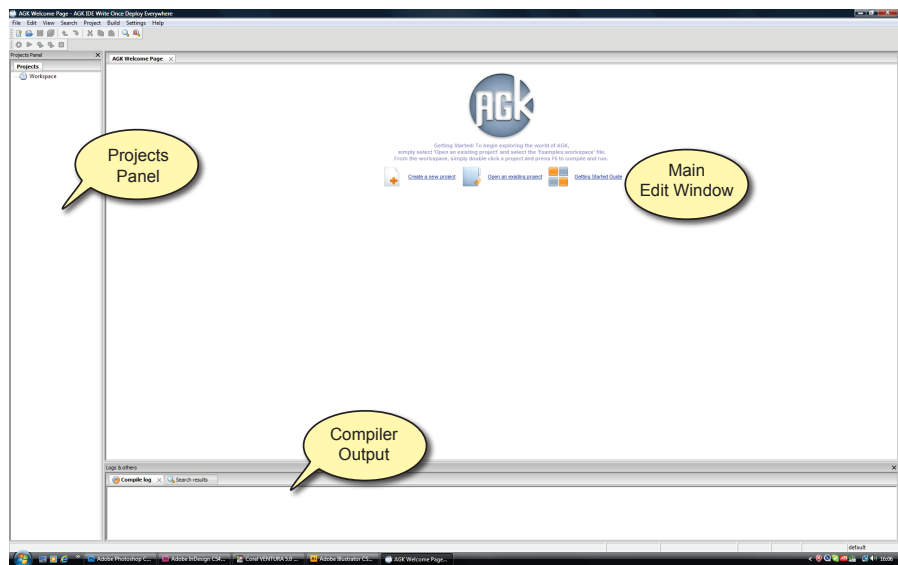


At the centre of the application window is the **Tip of the Day** window. If you don't want this to appear every time you start up AGK, just deselect the *Show tips at startup* check box. Once you close the **Tip of the Day** window, you are left with the three main areas of the AGK IDE (see FIG-2.3):

- | | | |
|-----------------------|---|--|
| The Main Edit Window | - | This where your program code is displayed once you start working on a project. |
| The Project Panel | - | This displays a tree structure of the files within the project(s) currently open. It only shows the names of those files containing code; the other files created by a project are not listed. |
| Compiler Output Panel | - | This panel (labelled as <i>Logs and others</i>) is used primarily to display information output by the compiler. |

FIG-2.3

AGK Layout



The steps required to create your first project are shown in FIG-2.4.

FIG-2.4

Creating a New Project

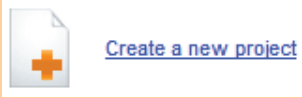
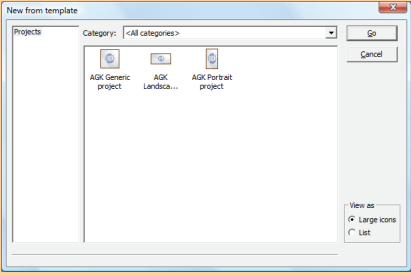
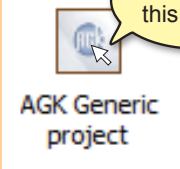
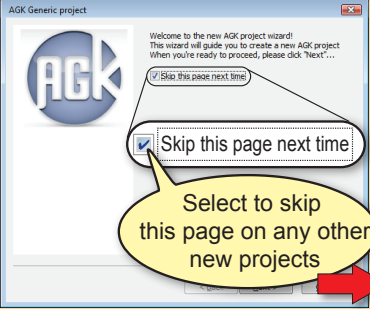
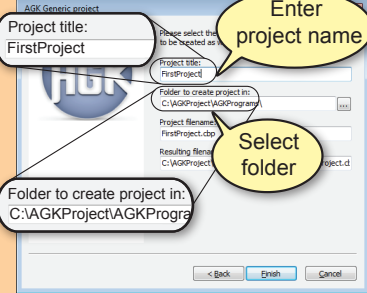
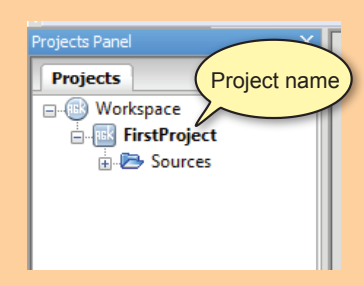
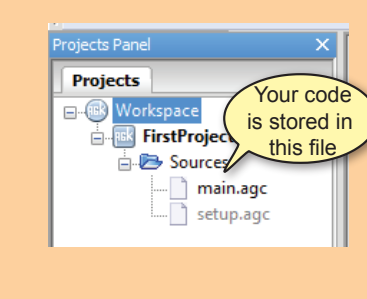
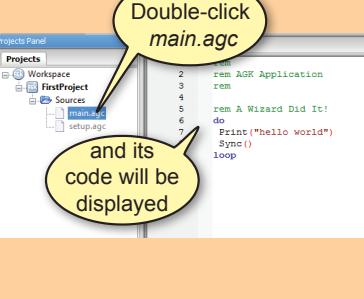

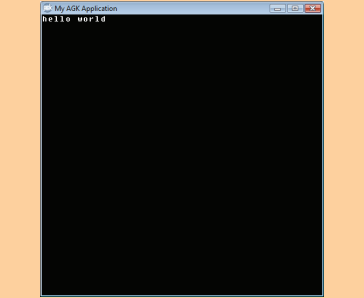
<p>Since this is our first project, we click on the Create a new Project option in the Main Edit Window (File New Project would work too).</p>	<p>This displays the Create from Template window which offers three different layout styles for your new project's display.</p>
	
<p>For this project, AGK Generic project is selected by double clicking that option.</p>	<p>This starts up the AGK project wizard. The first screen simply states that the wizard has started.</p>
 <p>Double-click this option</p>	 <p>Select to skip this page on any other new projects</p>

FIG-2.4
(continued)

Creating a New Project

➡ The project will create a new subfolder off the folder you select here. That subfolder will have the same name as your project.

<p>The second page of the wizard is where the project name and folder are selected. Other details are filled in automatically.</p>	<p>The Projects Panel now shows the new project and a folder called Sources.</p>
	
<p>Clicking on the Sources folder reveals the two source code files used by the project. main.agc will contain your code.</p>	<p>Double clicking on main.agc in the Project Panel opens its contents in a tabbed panel within the main edit window.</p>
	
<p>In fact, the AGK wizard has created a simple program within main.agc. This code can be run by pressing the Run button.</p>	<p>The sample program opens a small window to display its output.</p>
	

The new window created by running the sample program can be closed in the standard way by clicking on the X button at the top right.

This project will be used for the remaining programming activities in this chapter.

Activity 2.2

Before you start up AGK, create a main folder called *HandsOnAGK* on your disk drive. We'll use this as the main folder for all the AGK projects we are going to create throughout this book.

Load AGK then create, compile, and run your first project (named *FirstProject*) exactly as described in FIG-2.4, closing the app window once it has been run.

You may have noticed that the AGK software displayed messages in the compiler output area at the bottom of the screen (titled as *Logs & others*) to tell you that the app had been compiled and broadcast.

The Program Code

FIG-2.5 shows the code in *main.agk* that was automatically generated for us.

FIG-2.5

The Generated
Code

```
rem
rem AGK Application
rem
rem A Wizard Did It!
do
    Print("hello world")
    Sync()
loop
```

The line numbers that also appear in the edit window are not part of the code and are only there to help you identify the position of any line within the code.

Let's take a look at the code that was already generated for us and see what each of the lines means. The first lines are:

```
rem
rem AGK Application
rem

rem A Wizard Did It!
```

Blank lines and any lines starting with the term `rem` (short for REMARK) are treated as a comment by the compiler. Comments are there only for the benefit of us humans who happen to read the program code and are entirely ignored by the compiler when translating the instructions into machine code. Good comments will tell us the overall aim of the program as well as the purpose of individual sections of code. Comments can appear anywhere within a program.

```
do

loop
```

These two terms mark the start and end of an infinite loop - notice that no condition is given. Most AGK programs contain this loop which is designed to make sure all the code between these lines is continually executed until the user closes the app window. Without a loop of some type your program would start and finish so quickly that you would never have time to see what was displayed in the app window.

```
Print ("Hello world")
```

The `Print()` statement is used to state that some piece of information is to be displayed in the app window. The information itself is specified within a set of round brackets (more properly called **parentheses**). When that information contains letters (as opposed to numbers), then those letters must be enclosed in double quotes. Hence, the statement given above is an instruction to display the words *Hello world* on the screen. Note that the quotes themselves are not displayed.

```
Sync()
```

The `Sync()` statement is a command to update the contents of the app window. If you make any changes to what is displayed on the screen (for example, by executing a `Print()` statement), then you need to follow this with the statement `Sync()`. Without `Sync()` the screen display will not be updated.

Notice that the `Sync()` statement makes use of parentheses although no values are placed within them. However, omitting these parentheses would create a syntax error.

Activity 2.3

Change the `Print()` statement within *main.agc* so that the text enclosed in the double quotes reads *My first app*. This time click the **Compile** then **Run** buttons to compile and run the modified program. Was the new text displayed in the app window?

Select **File|Save** to save your modified program.



Compile Button



Run Button

Using the **Compile** button and then **Run** button separates the compilation and execution stages of the process into two distinct steps.

Running Your App to a Tablet or Smartphone

Although producing a true app for your smartphone or tablet is quite complex, you can, nevertheless, watch your app run on such a device. To do this, you need to first load the app AGK Player from the app store used by your device. For example, on an Android device, you will find AGK Player in Google Play.



Compile and
Broadcast Button

With AGK Player running on your target device and the app code you want to run on it loaded into AGK on your PC, press AGK's **Compile and Broadcast** button.

This will transfer the AGK program from the PC to your device through your WiFi setup. That means you either need to have a WiFi router attached to your PC or be using a laptop with built-in WiFi. The AGK Player app will detect your program being broadcast, download it, and run it on your device.

However, things are a bit more complicated if you have an Apple device. Apple won't support the AGK Player in their app store. As an alternative you can download the AGK Viewer from their store. The viewer isn't ideal but it will let you see a low-grade version of your app running on an Apple device. To run the AGK Player on your Apple device you will need to register as a developer. Details of how to do this are on the Game Creators' web site.

Activity 2.4

Make sure you have the AGK app player running on your device.

With the latest version of the project you created in Activity 2.3 showing on the AGK IDE, press the **Compile and Broadcast** button. Check that the program is now showing on your device.

Your program is not yet a true app - you can't save it on your device - it can only be executed using AGK Player. To create a true app for your device visit The Game Creators' web site for details.

First Statements in AGK BASIC

Introduction

Learning to program in AGK BASIC is very simple compared to other languages such as C++ or Java. Unlike most other programming languages, it has no rigid structure that the program itself must adhere to.

Now we need to start looking at the formal statements allowed in AGK BASIC and see how they can be used in a program.

Print()

We've already come across the `Print()` statement in our first program, so we already know that it is used to display information on the screen, but we need to know its exact format so that we don't create a syntax error by making a mistake in constructing the statement. The format of the `Print()` statement is shown in FIG-2.6.

FIG-2.6



`Print()`

This type of diagram is known as a **syntax diagram** for the obvious reason that it shows the syntax of the statement.

Each enclosed value in the diagram is known as a **token** (there are four tokens in the `Print()` statement). When you use a `Print()` statement in your program, its tokens must conform to those shown in the diagram. Some of the tokens must be an exact match for those in the diagram: **Print**, (, and) while others (only **value** in this case) have their actual value determined by the programmer.

Fixed values are shown in rounded-corners boxes, user-defined values are shown in regular boxes. In the case of the `Print()` statement, the term *value* is used to mean an integer value, a real value or a string value.

So, using the syntax diagram as a guide, we can see that the following are valid `Print()` statements:

```
Print("Hello world")
Print(12)
Print(0)
Print(-34.6)
```

while the following are not:

<code>Print 36</code>	(parentheses are missing)
<code>Print(Goodbye)</code>	(no quotes)
<code>Print('Help!')</code>	(single quotes used)

Activity 2.5

Which of the following are NOT valid `Print()` statements:

- a) `Print("-9.7")`
- b) `Print(0.0)`
- c) `Print(23, 51)`

Spaces

We can add spaces to a statement as long as those spaces do not split a single token into separate parts. So, for example, it is quite valid to write the line

```
Print      (    123  )
```

since each token can easily be identified, but

```
Pr      int ( 12    3 )
```

is not acceptable because the `Print` and `123` tokens have both been split into two parts.

Spaces can be omitted as long as doing so does not make it impossible to tell where one token ends and another begins. This is really only a problem when two or more adjacent tokens are constructed entirely from letters or numbers. So if we have a statement which begins with the code

```
if x = 3
```

then writing

```
ifx=3
```

would be invalid because the compiler would not be able to recognise the `if` and `x` as two separate tokens. On the other hand,

```
Print(123)
```

is correct because no adjacent tokens are constructed from alphanumeric characters.

Multiple Output

When we use two or more `Print()` statements, each value printed will be displayed on a separate line. For example, when the lines

```
Print("Hello")
Print("Goodbye")
```

are included in a program, they will create the output

```
Hello
Goodbye
```

Activity 2.6

Modify your program so that the main code now reads

```
do
    Print("First line")
    Print("Second line")
    Sync()
loop
```

Compile and run the program.

➡ Alphabetic and numeric characters are collectively known as **alphanumeric characters**.

You may want to save your project after each Activity by selecting

File|Save

Each message is on a separate line because the `Print()` statement always displays a new line character after the value specified and this causes the screen cursor to move to a new line.

Adding Comments

It is important that you add comments to any programs you write. These comments should explain the purpose of the program as a whole as well as what each section of code is doing. It's also good practice, when writing longer programs, to add comments giving details such as your name, date, programming language being used, hardware requirements of the program, and version number.

In AGK BASIC there are four ways to add comments:

FIG-2.7

rem

Add the keyword `rem`. The remainder of the line becomes a comment (see FIG-2.7).

rem text

FIG-2.8

Apostrophe
Comments



Add an apostrophe character (you'll find this on the top left key, just next to the 1). Again the remainder of the line is treated as a comment (see FIG-2.8).

' text

FIG-2.9

// Comments

Add two forward slashes followed by the descriptive text (see FIG-2.9).

// text

FIG-2.10

remstart..remend

Add several lines of comments by starting with the term `remstart` and ending with `remend`. Everything between these two words is treated as a comment (see FIG-2.10).

remstart
text
remend

This last diagram introduces another symbol - a looping arrowed line. This is used to indicate a section of the structure that may be repeated if required. In the diagram above it is used to signify that any number of comment lines can be placed between the `remstart` and `remend` keywords. For example, we can use this statement to create the following comment which contains three lines of text:

```
REMSTART
  This program is designed to play the game of
  battleships.
  Two peer-to-peer computers are required.
REMEND
```

PrintC()

The `PrintC()` statement is similar to `Print()` but does not add a new line character to the output. This means that each `PrintC()` statement's output is positioned on the screen immediately after the previous value. Hence,

```
PrintC("A")
```

```
PrintC("B")
```

would display AB

Activity 2.7

Change the two `Print()` statements in your program to `PrintC()` statements and observe the difference in output when the program is run.

Other Statements which Modify Output

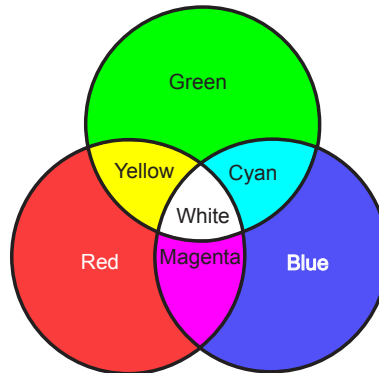
Other statements allow us to make various changes to how the information appearing on our screen is presented. We can change its colour, size, transparency and even the space between the characters.

Before we get started on instructions involving colour, perhaps it might be useful to go over a few basic facts about colour.

All colours you see on a monitor or TV are derived from the three primary colours red, green and blue. By varying the brightness of each of these three colours we can achieve almost any colour or shade the eye is capable of seeing. For example, mixing just red and green gives us yellow; blue and green gives us a colour called cyan, and blue and red gives magenta (see FIG-2.11).

FIG-2.11

Colours



Notice that all three colours together give white. The absence of all three colours gives black.

By varying the intensity (brightness) of each primary colour, we can create any shades or hues we require. AGK allows the intensity to vary between 0 (no colour) to 255 (full intensity). So pure white is achieved by setting all three colours to an intensity value of 255. For shades of grey, all three colours must have identical brightness values, but the lower that value, the darker the shade of grey.

SetPrintColor()

The `SetPrintColor()` sets the colour of all output created using the `Print()` and `PrintC()` statements. It can also be used to set the transparency of the text.

FIG-2.12

SetPrintColor()

```
SetPrintColor ( ( ired ( , igreen ( , iblue [ ( , itrans ] ) )
```

This syntax diagram introduces the use of square brackets. Tokens within square brackets are optional and can be omitted when using the statement.

In the above diagram:

➡ The value names start with i to indicate that integer values are required. Where a real number is needed, the value name will start with an f (for *float*). String values will start with an s.

ired	is an integer value giving the strength of the red component within the colour. This value should be in the range 0 to 255. 0 - no red; 255 - full red.
igreen	is an integer value (0 to 255) giving the strength of the green component.
ibblue	is an integer value (0 to 255) giving the strength of the blue component.
itrans	is an integer value (0 to 255) giving the amount of transparency. 0 - invisible, 255 - opaque.

Since the transparency value is optional and therefore can be omitted (in which case transparency stays at its current setting), we can use the statement simply to set the colour of any text being displayed by the `Print()` or `PrintC()` statements.

For example,

```
SetPrintColor(0,0,0)      rem *** sets text to black
SetPrintColor(255,255,255) rem *** sets text to white
SetPrintColor(255,0,0)    rem *** sets text to red
```

The `SetPrintColor()` statement must appear before the `Print()` or `PrintC()` statements whose output you wish to affect.

The statement only takes effect after a `Sync()` statement is executed.

Activity 2.8

Add a `SetPrintColor()` statement to your program, placing it immediately before your two `PrintC()` statements. Choose any colour values you wish.

Compile and run the program to check that the output is correct.

Once the colour has been set, all subsequent output will be in the specified colour. This means that there is no real need to place the `SetPrintColor()` statement inside the `do .. loop` structure where it will be executed every time the loop is repeated. Instead, that line of code can be moved to immediately before the `do` statement. Placed here, the statement will be performed only once, at the start of the program.

Activity 2.9

Reposition your `SetPrintColor()` statement, placing it on the line above `do`.

Compile and run the program again.

There should be no change to the output.

If there was no change to the output, what was the point of moving the statement?

The more lines of code that need to be executed, the slower a program runs. Let's say the statements within the loop are executed 200 times before you terminate the program. With the `SetPrintColor()` inside the loop, it would have been executed 200 times; with it outside the loop it is executed only once - so the program becomes more efficient.

If we include a value for *itrans* when we use `SetPrintColor()`, we can set the transparency of all text on the screen. The default transparency is 255, meaning the output is fully opaque. With a value of zero, the text would be invisible.

Activity 2.10

Modify the `SetPrintColor()` statement in your program, adding 126 as the transparency value.

Run the program and see what effect the changes have made to the output.

Try other transparency values to see their effect.

SetPrintSize()

The `SetPrintSize()` statement (see FIG-2.13) sets the size of the text displayed by a `Print()` or `PrintC()` statement.

FIG-2.13

SetPrintSize()

`SetPrintSize` ((`size`))

where:

size is a real number setting the size of characters. The default value for characters is about 3.5.

Activity 2.11

Add the line

```
SetPrintSize(8.6)
```

immediately after your `SetPrintColor()` statement (reset the transparency value to 255).

Compile and run the program. What do you notice about the quality of the text produced?

The reason that the text seems blurred when it is enlarged is that the text itself is stored as an image. Enlarging that image causes blurring.

SetPrintSpacing()

This statement (see FIG-2.14) adjusts the spacing between the characters shown on the screen.

FIG-2.14

SetPrintSpacing()

`SetPrintSpacing` ((`gap`))

where:

gap is a real number giving the gap between characters. The default

is zero. Larger values widen the gap; negative values cause the gap to decrease and even to make letters overlap.

Activity 2.12

Add a `SetPrintSpacing()` statement to your program, placing it before the `do .. loop` structure. Set the gap size to 5.5.

Compile and run the program to check how the output is changed.

Change the value used to -2.5 and observe the effect on the output.

Message()

Another way of displaying text on the screen is to use the `Message()` statement. This creates a more prominent output, placing the text in a separate window. The format of the `Message()` statement is shown in FIG-2.15.

FIG-2.15

`Message()`

`Message (stext)`

where

stext is a string containing the message to be displayed.

For example, the line

```
Message("hello world")
```

produces the output shown in FIG-2.16 when run on a PC.

FIG-2.16

A Typical
Message Window



The exact style of the window produced depends on the device on which your app is being run.

SetClearColor()

You will have noticed that the window created by your AGK app always has a black background. This default color can be changed using the `SetClearColor()` statement which has the format shown in FIG-2.17.

FIG-2.17

`SetClearColor()`

`SetClearColor (ired , igreen , iblue)`

where:

ired is an integer value (0 to 255) giving the strength of the red component.

igreen is an integer value (0 to 255) giving the strength of the green component.

ibblue is an integer value (0 to 255) giving the strength of the blue component.

ClearScreen()

The `SetClearColor()` statement only works when followed by a `Sync()` or a `ClearScreen()` statement which has the same effect. The format for the `ClearScreen()` statement is given in FIG-2.18.

FIG-2.18

ClearScreen()

ClearScreen (())

So to create a yellow background on the screen, we would start our program with the lines:

```
SetClearColor()  
ClearScreen()
```

Often this statement will appear at the start of a program, but you may wish to change the colour at a later stage perhaps to indicate that a game has entered a new phase.

Activity 2.13

Change the background of the app window to red and test your program.

Positioning the Print() Statements

We have placed the various statements affecting the colour, size and spacing of our text before the `do..loop` structure on the basis that these commands need only be performed once. So you may be tempted to think that surely we can do the same thing with the `Print()` and `Sync()` statements since the displayed text remains unchanged throughout the running of the program. Let's see what happens when we try this.

As you can see from the output produced, for a simple program such as this, moving the statements has had no effect on the output produced. We are left with an empty `do..loop` which makes sure that the program does not terminate before we click the app window's close button.

Activity 2.14

Move the `PrintC()` and `Sync()` statements in your program so that they are positioned immediately before the `do` statement.

What effect does this have when you run your program?

Although we now know that it is possible to place the `Print()` and `Sync()` statements outside the `do` loop it is usually not a good idea to do so in any but the simplest programs since it can create other problems which we will discuss in a later chapter.

Summary

- Programs are written using a programming language.
- Programming language code must be translated into machine code before the program can be executed by the computer.
- The stored program code is known as the source file; the stored machine code

as the object file.

- Each line of a program must conform to the rules of syntax.
- An error in how a line is written is known as a syntax error.
- AGK programs can be written in BASIC or C++.
- The collection of files created when writing an AGK app is known as a project.
- The main file in an AGK project is *main.agc* which contains the program code.
- The AGK development package is an Integrated Development Environment.
This allows edit, compiling and testing to be performed from within the same program.
- To download an app to your digital device, the player must be installed and running on that device and the app broadcast from the AGK IDE.
- When an app is being tested it creates an app window.
- Comments can be added to your code using `rem`, ```, or `remstart..remend`.
- Comments help us understand the purpose of a piece of code but are ignored by the compiler.
- Use `Print()` to display information on the screen.
- Use `PrintC()` to display information without moving to a new line afterwards.
- Use `SetPrintColor()` to set the colour used when displaying text.
- Use `SetPrintSize()` to set the size of future text output.
- Use `SetPrintSpacing()` to set the spacing between characters in future text output.
- Use `Message()` to display a message in a separate window.
- Use `SetClearColor()` to set a background colour for the app screen.

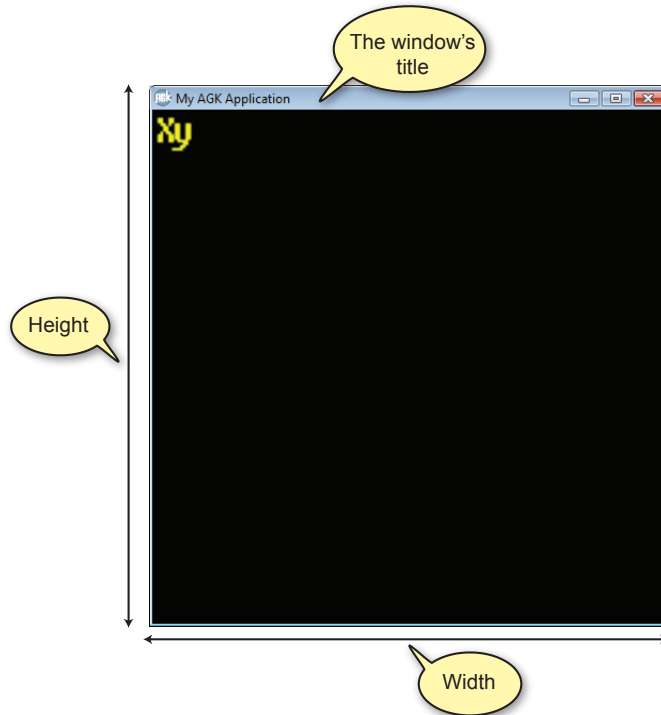
The Second Source File

Every project you create actually contains a second *.agc* file. You can see it listed in the Projects Panel immediately below *main.agc*.

Although you are not free to add lines of code to this file as you can with *main.agc*, you are allowed to change the values given. Those values determine the title and dimensions of the window in which your app appears when run under Microsoft Windows. For example, the window of a typical program (see FIG-2.16) reflects the details given in *setup.agc*.

FIG-2.16

The App Window



By changing the values specified in the first three lines of *setup.agc* (ignoring the *rem* lines), we can change the characteristics of the window.

Activity 2.15

Double click *setup.agc* in the Projects Panel to display its code. Change the appropriate existing lines to read:

```
title=My First App
width=320
height=480
```

Make sure the only spaces with these lines are those in the title.

Compile and run your program to see what changes this has made.

These characteristics given in *setup.agc* only affect the layout of the window on your PC. Other statements (covered later) need to be included in your program to set the app screen size on a tablet or phone.

A Splash Screen

A common feature of many games is a **splash screen**. A splash screen is simply a graphic that displays for a few moments at the start of the game. Typically a splash screen will contain an image giving the flavour of the game play that is about to follow as well as the name of the game and the publishing company.

AGK allows you to add a splash screen to your game without any coding whatsoever.

If you load Windows Explorer and have a look in the folder created by AGK to hold the files belonging to your project (*HandsOnAGK/FirstProject*), you should see contents similar to that shown in FIG-2.17.

FIG-2.17

AGK Project's
Files

Name	Date modified	Type	Size
media	19/07/2011 07:50	File Folder	
FirstProject.byc	19/07/2011 07:50	BVC File	2 KB
FirstProject.cbp	17/07/2011 15:53	CBP File	1 KB
FirstProject.dbpro	19/07/2011 07:50	DBPRO File	1 KB
FirstProject.exe	08/07/2011 14:55	Application	1,099 KB
FirstProject.layout	19/07/2011 07:59	LAYOUT File	1 KB
main.agc	19/07/2011 07:50	AGC File	1 KB
setup.agc	27/06/2011 17:16	AGC File	1 KB

The splash screen graphic file must be placed in the project's main folder. The file must be in **PNG** format and be called *AGKSplash.png*. No other name is acceptable. The image is best set to the same size as the window dimensions (in our case, 480 x 320). An example of a splash screen is shown in FIG-2.18.

FIG-2.18

A Splash Screen



Rather than create your own image, you can use the one supplied in the downloads that accompany this book.

Activity 2.16

Open a paint program you have available and create a 480 pixels high by 320 pixels wide image. Save the file in PNG format in the folder *HandsOnAGK/FirstProject* naming the file *AGKSplash.png*.

In AGK, recompile your program and run it. You should see your splash screen appear when the app window first opens.

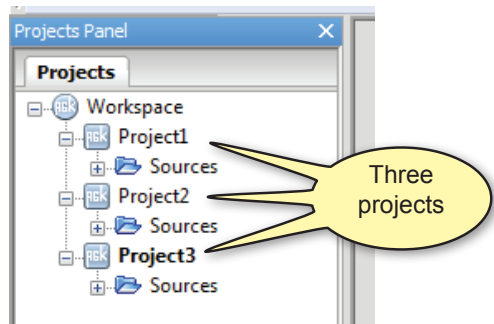
Starting a New Project

When you first start up AGK for a work session, we've already seen that it will give you the option to create a new project. Should you want to create more new projects during that session, you can do so from the main menu (**File|New|Project**).

However, the Projects Panel will display all of the projects you have been using (see FIG-2.19).

FIG-2.19

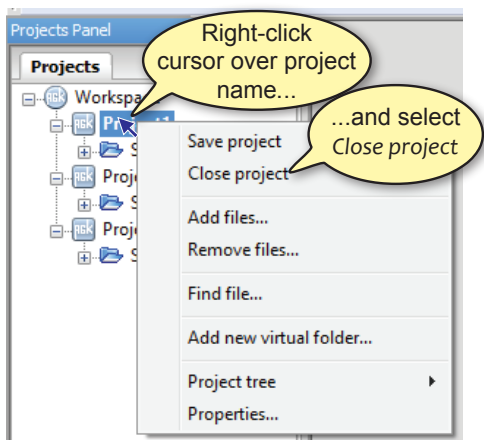
Multiple Projects



Having several projects open at the same time can be a bit confusing when you first start using AGK, so the best option is to close projects that you are not currently working on. FIG-2.20 shows how to close a project from the Projects Panel.

FIG-2.20

Multiple Projects



From now on, make sure you always close any old project before starting a new one.

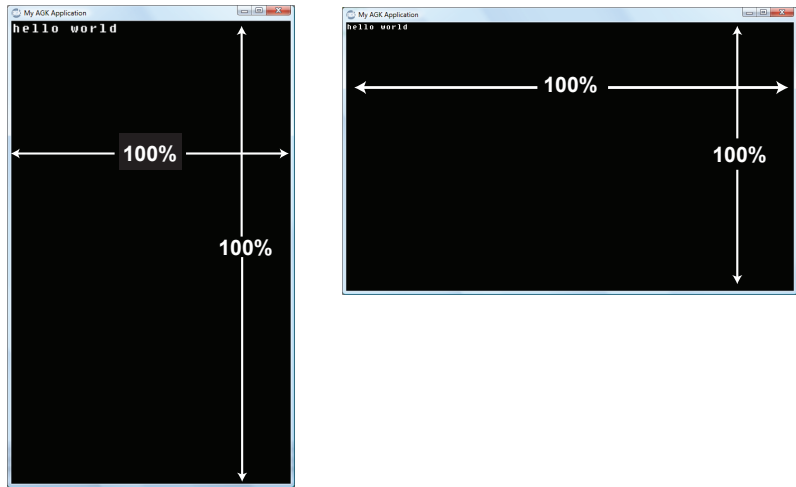
App Window Properties

Measurements

By default, AGK apps use a percentage measurement system. This means that no matter the actual dimensions of the app window, AGK always treats the width as 100% and the height as 100% (see FIG-2.21).

FIG-2.21

The Screen's Percentage Measurement System



When you want to position an item on the screen it is done using percentage measurements. For example, the position (50,50) represents the middle of the app window irrespective of the window's actual dimensions.

Percentage values are also used when setting the size of various visual elements. For example, earlier in this chapter we made use of the `SetPrintSize()` statement to resize the text created by any subsequent `Print()` statement. The value supplied to this statement represents the high of the text as a percentage of the screen height. Of course, this means that text set to a height of 4 will appear taller in a long window and smaller in a short window. In fact, you can see that in the “Hello world” text visible in FIG-2.21 above.

All programs in this book use the default percentage system.

SetDisplayAspect()

When using the percentage measuring system, the *setup.agc* file is used to set the actual size of the app window on your PC, but if you intend to transfer that app to another device such as a smartphone or tablet, you should explicitly set the aspect ratio (width to height) using the `SetDisplayAspect()` statement (see FIG-2.22).

FIG-2.22

SetDisplayAspect()

`SetDisplayAspect (ratio)`

where:

ratio is a real number giving the width to height ratio. For example, iPhone and iPad have an aspect ratio of 4.0/3.0 (1.3333).

Use zero as the *ratio* value if you want the width and height values in the *setup.agc* file to be used to determine the aspect ratio. Use -1 if you want the app to fill the whole screen irrespective of aspect ratio. Using this last option may distort visual

elements of the app if the device's aspect ratio is different to that used when developing the app (like watching an old 4/3 TV program on your widescreen TV).

SetVirtualResolution()

If you would rather work with a resolution based on pixels, have your program execute the `SetVirtualResolution()` statement when it starts up. The statement's format is shown in FIG-2.23.

FIG-2.23

SetVirtualResolution()

`SetVirtualResolution ((iwidth , iheight)`

where:

iwidth is an integer value giving the nominal width of the app window in pixels.

iheight is an integer value giving the nominal height of the app window in pixels.

If you were writing an app for the original iPhone, you would set the resolution to 320×480 using the line:

```
SetVirtualResolution(320,480)
```

When you are developing your app on your PC, the app window will take on the actual size specified in the `SetVirtualResolution()` statement. However, when you transfer the app to another device, the app will expand (or contract) to fit that device's screen. For example, if you run your 320×480 app on a newer iPhone with its 640×960 resolution, your AGK will automatically expand to fill the full screen.

This is why the term **virtual resolution** is used; this development resolution may in fact be different from the actual resolution used when the app is running on a device other than your PC.

The only problem arises when the device on which your app is running has a different aspect ration (width / height) than that specified in the `SetVirtualResolution()` statement. Expanding the app's resolution from 320×480 to 640×960 isn't a problem because both have an aspect ratio of 3/4. But if we were to try and run the same app on an original Asus EEE Transformer which has a resolution of 1280×800 (an aspect ratio of 8/5) then we have a problem. Expanding the app to fill a 8/5 screen would cause distortion of any images being displayed (circles would become ovals!). AGK handles this by creating as large a 3/4 ratio images as possible and adding a border to the remainder of the screen.

When you use `SetVirtualResolution()` in your app, all screen positions and sizes are given in **virtual pixels**.

SetBorderColor()

You can specify the border colour to be used when you app runs on a device with a different aspect ratio to that specified in the app's code using the `SetBorderColor()` statement (see FIG-2.24).

FIG-2.24

SetBorderColor()

`SetBorderColor ((ired , igreen , iblue)`

where:

ired	is an integer variable (0 to 255) giving the intensity of the red component of the border colour to be used. 0: no red; 255: full red.
igreen	is an integer variable (0 to 255) giving the intensity of the green component of the border colour. 0: no green; 255: full green.
ibblue	is an integer variable (0 to 255) giving the intensity of the blue component of the border colour. 0: no blue; 255: full blue.

To create a grey border we could use a statement such as:

```
SetBorderColor(120,120,120)
```

SetWindowTitle()

For apps that are running in a windows based environment (on PCs or Macs), you can set the title that appears at the top of the window using the `SetWindowTitle()` statement (see FIG-2.25).

FIG-2.25

SetWindowTitle()



```
SetWindowTitle ( stext )
```

where

stext is a sting containing the text to appear in the window title bar.

A typical statement would be:

```
SetWindowTitle("Jigsaw Game")
```

Further screen-handling statements are covered in Chapter 19.

Summary

- By default, AGK uses a percentage coordinate system within the app window.
- Use `SetVirtualResolution()` to use a virtual pixel coordinate system.
- Use `SetDisplayAspect()` to set the width to height ratio of the screen/window.
- Use `SetBorderColor()` to specify a colour for any part of the physical screen ot included in the app's output area.
- Use `SetWindowTitle()` to specify a title for any windows-based app.

Solutions

Activity 2.1

- Machine code instruction. These are stored as a sequence of binary digits.
- A compiler.
- A syntax error.

Activity 2.2

No solution required.

Activity 2.3

Your code should now read (`rem` statements have been omitted):

```
do
  Print("My first app")
  Sync()
loop
```

Compile and run your code.

The new text should be displayed in the app window when the program is run.

Select **File|Save**

Activity 2.4

Activity 2.5

- Valid. Any characters can be enclosed in quotes - including numeric ones.
- Valid. A real number.
- Invalid. Only a single value can be displayed.

Activity 2.6

Your program code should be:

```
do
  Print("First line")
  Print("Second line")
  Sync()
loop
```

The output should be:

```
First line
Second line
```

Activity 2.7

Program code:

```
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

The output should be:

```
First lineSecond line
```

If you want a space between the two outputs, you would need to include a space inside the quotes at the end of the first piece of text or at the start of the second.

Activity 2.8

Program code (your colour values will be different):

```
do
  SetPrintColor(255,255,0) rem *** yellow ***
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

Activity 2.9

Program code (your colour values will be different):

```
SetPrintColor(255,255,0) rem *** yellow ***
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

Activity 2.10

Program code (your colour values will be different):

```
SetPrintColor(255,255,0,126) rem *** yellow ***
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

The text output will appear darker as the black background shows through.

Activity 2.11

Program code (your colour values will be different):

```
SetPrintColor(255,255,0,126) rem *** yellow ***
SetPrintSize(8.6)
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

The text will appear larger but somewhat blurred.

Activity 2.12

Program code (your colour values will be different):

```
SetPrintColor(255,255,0,126) rem *** yellow ***
SetPrintSize(8.6)
SetPrintSpacing(5.5)
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

The characters in the output text will be widely spaced.

The `SetPrintSpacing()` line should then be changed to

```
SetPrintSpacing(-2.5)
```

The characters will now bunch together.

Activity 2.13

Program code:

```
SetClearColor(255,0,0)
ClearScreen()
SetPrintColor(255,255,0,126) rem *** yellow ***
SetPrintSize(8.6)
SetPrintSpacing(-2.5)
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

Activity 2.14

Program code:

```
SetClearColor(255,0,0)
ClearScreen()
SetPrintColor(255,255,0,126) rem *** yellow ***
SetPrintSize(8.6)
SetPrintSpacing(-2.5)
PrintC("First line")
PrintC("Second line")
Sync()
do
loop
```

The output remains unchanged.

Activity 2.15

The app window title and dimensions should be changed.

Activity 2.16

No solution required.

3

Data

In this Chapter:

- ☐ Constants
- ☐ Variables
- ☐ Naming Variables
- ☐ Assigning Values to Variables
- ☐ Arithmetic Operators
- ☐ Operator Precedence
- ☐ Random Numbers
- ☐ Determining the Elapsed Time

Program Data

Introduction

Every computer game has to store and manipulate facts and figures (more commonly known as **data**). For example, a program may store the name of a player, the number of lives remaining or the time left in which to complete a task.

We've already seen that all basic data can be grouped into three basic types:

Real values are also known as **floating-point** or simply **float** values.

integer	-	any whole number, positive, negative or zero
real	-	any number containing a decimal point
string	-	any collection of characters (may include numeric characters)

For example, if player *Ian Knot* had 3 lives and 10.6 minutes to complete a game, then:

3	is an example of an integer value
10.6	is a real value
Ian Knot	is an example of a string

Activity 3.1

Identify the type of value for each of the following :

- | | | | |
|--------------|----------|---------|-------------|
| a) -9 | b) abc | c) 18 | d) 12.8 |
| e) ? | f) 0 | g) -3.0 | h) Mary had |
| i) 4 minutes | j) 0.023 | | |

Constants

When a specific value appears in a computer program's code it is usually referred to as a **constant**. Hence, in the statement

```
Print(7)
```

the value 7 is a constant. When identifying a value as a constant, the constant's type is often included in the description, so, for example, 7 is an **integer constant**.

Activity 3.2

What type of constants are the following:

- | | | | |
|--------|--------------|---------|---------|
| a) -12 | b) Elizabeth | c) 3.14 | d) 27.0 |
|--------|--------------|---------|---------|

Variables

Most programs not only need to display information, but also need to store data and calculate results. To store data in AGK BASIC we need to use a **variable**. A variable is, in effect, reserved space within the computer's memory where a single value can be stored. Every variable in a program is assigned a unique name and can store only a single value. When a variable is first created, the type of value it can store (integer, real or string) is specified. No other type of value can be stored in that variable. For

example, a variable designed to store an integer value cannot store a string.

Integer Variables

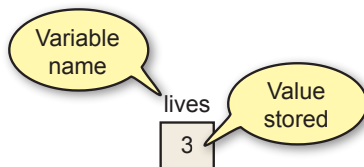
In AGK BASIC variables are created automatically as soon as we mention them in our code. For example, let's assume we want to store the number of lives allocated to a game player in a variable called *lives*. To do this in AGK BASIC we simply write the line:

```
lives = 3
```

This sets up a variable called *lives* in the computer's memory and stores the value 3 in that variable (see FIG-3.1)

FIG-3.1

Storing Data in a Variable



This instruction is known as an **assignment statement** since we are assigning a value (3) to a variable (*lives*).

You are free to change the contents of a variable at any time by assigning it a different value. For example, we can change the contents of *lives* with a line such as:

```
lives = 2
```

When we do this, any previous value will be removed and the new value stored in its place (see FIG-3.2).

FIG-3.2

Changing the Value in a Variable



The variable *lives* is designed to store an integer value. In the lines below, *a*, *b*, *c*, *d*, and *e* are also integer variables. So the following assignments are correct

```
a = 200
b = 0
c = -8
```

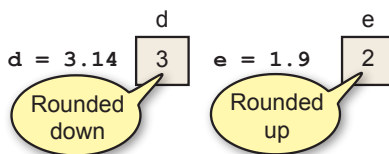
but the lines below will cause problems

```
d = 3.14
e = 1.9
```

since they attempt to store real constants in variables designed to hold integers. AGK BASIC won't actually report an error if you try out these last two examples, it simply rounds the fractional part of the numbers and ends up storing 3 in *d* and 2 in *e* (see FIG-3.3). Fractions of 0.5 and above are rounded up, other values are rounded down.

FIG-3.3

Integer Variables Round Real Values



Real Variables

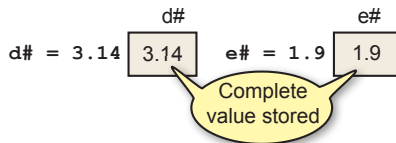
If you want to create a variable capable of storing a real number, then you must end the variable name with the hash (#) symbol. For example, if we write

```
d# = 3.14
e# = -1.9
```

we have created variables named *d#* and *e#*, both capable of storing real values (see FIG-3.4).

FIG-3.4

Real Variables



Any number (real or integer) can be assigned to a real variable, so we could write a statement such as:

```
d# = 12
```

Although we may assign an integer to a real variable, the value will be stored as a real. Therefore, when the statement above has been executed, *d#* will contain 12.0.

If any numeric value can be stored in a real variable, why bother with integer variables? Actually, you should always use integer values wherever possible because some hardware can be much faster at handling integer values than real ones. Also, real numbers can be slightly inaccurate because of rounding errors within the machine. For example, the value 2.3 might be stored as 2.2999987. Another consideration is that a real variable requires more space in the computer's memory than an integer one.

String Variables

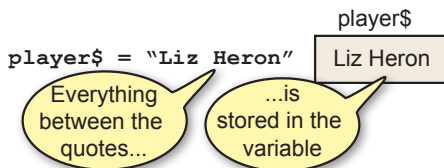
Finally, if you want to store a string value, you need to use a string variable. String variable names must end with a dollar (\$) sign. The value to be stored must be enclosed in double quotes. We could create a string variable named *player\$* and store the name *Liz Heron* in it using the statement:

```
player$ = "Liz Heron"
```

The double quotes are not stored in the variable (see FIG-3.5).

FIG-3.5

String Variables



Absolutely any value can be stored in a string variable as long as that value is enclosed in double quotes. Below are a few examples:

```
a$ = ">%"
b$ = "Your spaceship has been destroyed"
c$ = "That costs $12.50"
d$ = ""      rem *** A string containing no characters ***
```


Activity 3.3

Which of the following are valid AGK BASIC statements that will store the specified value in the named variable?

a) `a = 6`
d) `d$ = 5`

b) `b = 12.89`
e) `e$ = 'Goodbye'`

c) `c = "Hello"`
f) `f# = -12.5`

Using Meaningful Names

It is important that you use meaningful names for your variables when you write a program. This helps you remember what a variable is being used for when you go back and look at your program a month or two after you wrote it. So, rather than write statements such as

```
a = 3
b = 120
c = 2000
```

a better set of statements would be

```
lives = 3
points = 120
timeremaining = 2000
```

which give a much clearer indication of the purpose of the variables.

Naming Rules

AGK BASIC, like all other programming languages, demands that you follow a few rules when you make up a variable name. These rules are:

- The name should start with a letter.
- Subsequent characters in the name can be a letter, number, or underscore.
- The final character can be a # (needed when creating real variables) or \$ (needed when creating string variables).
- Upper or lower case letters can be used, but such differences are ignored. Hence, the terms *total* and *TOTAL* refer to the same variable.
- The name cannot be an AGK BASIC keyword.

This means that variable names such as

```
a, bc, de_2, fgh$, iJKlmnp#
```

are valid, while names such as

```
2a, time-remaining
```

are invalid.

The most common mistake people make is to have a space in their variable names (e.g. *fuel level*). This is not allowed. As a valid alternative, you can replace the space with an underscore (*fuel_level*) or join the words together (*fuellevel*). Using capital

➡ A keyword is any term that is used as part of the language. For example, if, then, for, repeat, etc.

2a - cannot start with a numeric digit.

time-remaining - hyphen not allowed.

letters for the joined words is also popular (*FuelLevel*).

Note that the names *no*, *no#* and *no\$* represent three different variables; one designed to hold an integer value (*no*), one a real value (*no#*) and the last a string (*no\$*).

Activity 3.4

Which of the following are invalid variable names:

- | | | |
|----------|----------------|------------|
| a) x | b) 5 | c) "total" |
| d) al2\$ | e) total score | f) ts#o |
| g) then | h) G2_F3 | |

Named Constants

We have already seen that assigning meaningful names to the variables used in a program aids readability. When a program uses a fixed value which has an important role within the program (for example, perhaps the value 1000 is the score a player must achieve to win a game), then we have the option of assigning a name to that value using the `#constant` statement. The format of the `#constant` statement is shown in FIG-3.6.

FIG-3.6

`#constant` `name` `[=]` `value`

`#constant`

where:

- | | |
|--------------|---|
| name | is the name to be assigned to the constant value. A common convention is to assign an uppercase name making it easy to distinguish between variable names and constant names. |
| value | is the constant value being named. |

For example, we can name the value 1000 WINNINGSCORE using the line:

```
#constant WINNINGSCORE = 1000
```

Since the equal sign (=) is optional, it is also valid to write:

```
#constant WINNINGSCORE 1000
```

Real and string constants can also be named, but the names assigned must NOT end with # or \$ symbols. Therefore the following lines are valid:

```
#constant PASSWORD = "neno"  
#constant PI 3.14159
```

The value assigned to a name cannot be changed, so having written

```
#constant WINNINGSCORE = 1000
```

it is not valid to try to assign a new value later in the program with a line such as:

```
WINNINGSCORE = 1900
```

The two main reasons for using named constants in a program are:

- 1) Aiding the readability of the program. For example, it is easier to understand the meaning of the line

```
if playerscore >= WINNINGSCORE
```

than

```
if playerscore >= 1000
```

- 2) If the same constant value is used in several places throughout a program, it is easier to change its value if it is defined as a named constant. For example, if, when writing a second version of a game we decide that the winning score has to be changed from 1000 to 2000, then we need only change the line

```
#constant WINNINGSCORE = 1000
```

to

```
#constant WINNINGSCORE = 2000
```

On the other hand, if we've used lines such as

```
if playerscore >= 1000
```

throughout our program, every one of those lines will have to be changed so that the value within them is changed from 1000 to 2000.

Summary

- Fixed values are known as constants.
- There are three types of constants: integer, real and string.
- String constants are always enclosed in double quotes.
- The double quotes are not part of the string constant.
- A variable is a space within the computer's memory where a value can be stored.
- Every variable must have a name.
- A variable's name determines which type of value it may hold.
- Variables that end with the # symbol can hold real values.
- Variables that end with the \$ symbol can hold string values.
- Other variables hold integer values.
- The name given to a variable should reflect the value held in that variable.
- When naming a variable the following rules apply:
 - The name must start with a letter.
 - Subsequent characters in the name can be numeric, alphabetic or the underscore character.
 - The name may end with a # or \$ symbol.
 - The name must not be an AGK BASIC keyword.
- Constants can also be assigned a name.

Allocating Values to Variables

Introduction

There are several ways to place a value in a variable. Some of the AGK BASIC statements available to achieve this are described below.

The Assignment Statement

In the last few pages we've used AGK BASIC's assignment statement to store a value in a variable. This statement allows the programmer to place a specific value in a variable, or to store the result of some calculation.

The assignment statement has the form shown in FIG-3.7.

FIG-3.7

The Assignment Statement



```
variable = value
```

The value copied into the variable may be one of the following:

- a constant
- the contents of another variable
- the result of an arithmetic expression

Examples of each are shown below.

Assigning a Constant

This is the type of assignment we've seen earlier, with examples such as

```
name$ = "Liz Heron"
```

where a fixed value (a constant) is copied into the variable. As a general rule, make sure that the value being assigned is of the same data type as the variable. However, an integer value may be copied into a real variable, as in the line:

```
result# = 33
```

The program deals with this by storing the value assigned to *result#* as 33.0.

Activity 3.5

What are the minimum changes required to make the following statements operate correctly?

a) `desc = "tail"`

b) `result = 12.34`

If you try copying a real value to an integer variable, the real value will be rounded to the nearest integer and that value stored in the variable. Hence, the line

```
number = 33.5
```

will result in the value 34 being stored in *number* (value rounded up), while the assignment

```
result = 12.2
```

will store 12 in *result* (value rounded down).

Copying Another Variable's Value

Once we've assigned a value to a variable in a statement such as

```
no1 = 12
```

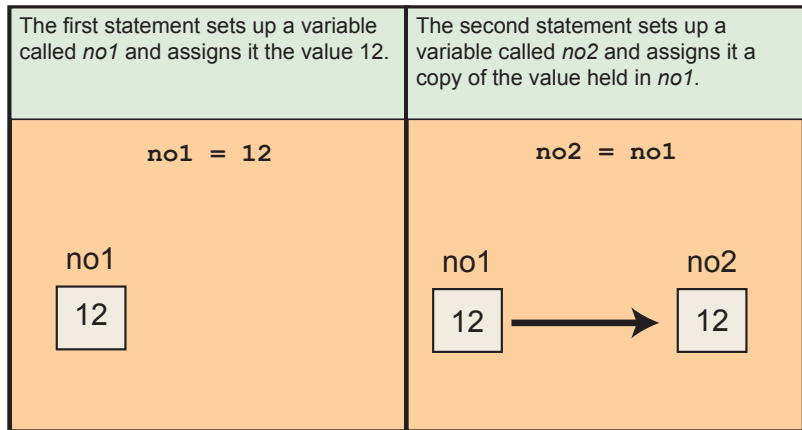
we can copy the contents of that variable into another variable with the command:

```
no2 = no1
```

The effect of these two statements is shown in FIG-3.8.

FIG-3.8

Copying from
Another Variable



When the assignment is complete, both variables will contain the value 12. As before, you must make sure the two variables are of the same type, although the contents of an integer variable may be copied to a real variable as in the line:

```
ans# = no1
```

Copying the contents of a real variable to an integer variable will cause rounding to the nearest integer. For example,

```
ans# = -12.94  
no1 = ans#
```

will store -13 in *no1*.

Activity 3.6

Assuming a program starts with the lines:

```
no1 = 23  
weight# = 125.8  
description$ = "sword"
```

which of the following instructions would be invalid?

- | | | |
|----------------------------|---------------------------------|--|
| a) <code>no2 = no1</code> | b) <code>no3 = weight#</code> | c) <code>result = description\$</code> |
| d) <code>ans# = no1</code> | e) <code>abc\$ = weight#</code> | f) <code>m# = description\$</code> |

Assigning the Result of an Arithmetic Expression

Another variation for the assignment statement is to have it perform a calculation and then store the result of that calculation in the named variable. Hence, we might write

```
no1 = 7 + 3
```

which would store the value 10 in the variable *no1*.

The example shows the use of the addition operator, but there are 6 possible operators that may be used when performing a calculation. These are shown in FIG-3.9.

FIG-3.9

Arithmetic
Operators

Operator	Function	Example
+	addition	no1 = no2 + 5
-	subtraction	no1 = no2 - 9
*	multiplication	ans = no1 * no2
/	division	r1# = no1/ 2.0
mod	remainder	ans = no2 mod 3
^	power	ans = 2^3

The result of most statements should be obvious. For example, if a program begins with the statements

```
no1 = 12
no2 = 3
```

and then contains the line

```
total = no1 - no2
```

then the variable *total* will contain the value 9, while the line

```
product = no1 * no2
```

stores the value 36 in the variable *product*.

The remainder operator (**mod**) is used to find the integer remainder after dividing one integer into another. For example,

```
ans = 9 mod 5
```

assigns the value 4 to the variable *ans* since 5 divides into 9 once with a remainder of 4. Other examples are given below:

```
6 mod 3      gives 0
7 mod 9      gives 7
123 mod 10   gives 3
```

If the first value is negative, then any remainder is also negative:

```
-11 mod 3    gives -2
```

Activity 3.7

What is the result of the following calculations:

- a) `12 mod 5` b) `-7 mod 2` c) `5 mod 11` d) `-12 mod -8`

The power operator (\wedge) allows us to perform a calculation of the form x^y . For example, a 24-bit address bus on the microprocessor of your computer allows 2^{24} memory addresses. We could calculate this number with the statement:

```
addresses = 2^24
```

Most of the results produced by these operators are easy to calculate manually as long as you are capable of basic arithmetic. However, the results of some statements are not quite so obvious. For example, you might expect the line

```
ans# = 19/4
```

to store the value 4.75 in *ans#*. In fact, the value stored will be 4.0. This is because the division operator always returns an integer result if the two values involved are both integer. On the other hand, if we write

```
ans# = 19/4.0
```

and thereby use a real value in the calculation, then the result stored in *ans#* will be the expected 4.75.

When using the division operator, a situation that you must guard against is division by zero. In mathematics, dividing any number by zero gives an undefined result, so most programming languages get quite upset if you try to get them to perform such a calculation. AGK BASIC, on the other hand, will, when presented with a line such as

```
ans = 10/0
```

store the value 0 in *ans*.

You might be tempted to think that you would never write such a statement, but a more likely scenario is that your program contains a line such as

```
ans = no1 / no2
```

and if *no2* contains the value zero, attempting to execute the line will still cause a value of zero to be stored in *ans*.

Some statements may not appear to make sense if you are used to traditional algebra. For example, what is the meaning of a line such as

```
no1 = no1 + 3
```

In fact, it means add 3 to *no1*. We can take the literal meaning of the statement to be:

Take the value currently stored in no1, add 3, and store the result back in no1.

Another unusual assignment statement is of the form:

```
no1 = -no1
```

The effect of this statement is to change the sign of the value held in *no1*. For example, if *no1* contained the value 12, the above statement would change that value to -12. Alternatively, if *no1* started off containing the value -12, the above statement would change *no1*'s contents to 12.

Activity 3.8

Assuming a program starts with the lines:

```
no1 = 2
v# = 41.09
```

what will be the result of the following instructions?

- | | | |
|-------------------------------|----------------------------|--------------------------------|
| a) <code>no2 = no1^4</code> | b) <code>x# = v#*2</code> | c) <code>no3 = no1/5</code> |
| d) <code>no4 = no1 + 7</code> | e) <code>m# = no1/5</code> | f) <code>v2# = v# - 0.1</code> |
| g) <code>no1 = no1 + 1</code> | h) <code>no5 = -no1</code> | |

Treat each statement separately - don't assume the results are cumulative.

The inc and dec Statements

Because adding to or subtract from the existing value in a variable is so common, AGK BASIC has added statements specifically to perform those tasks.

The `inc` statement (short for *increment*) allows you to add 1 or any other value to the current contents of a variable. So rather than write

```
no1 = no1 + 1
```

we can write

```
inc no1
```

and in place of

```
num = num + 7
```

we can write

```
inc num, 7
```

Note that no value needs to be given when 1 is being added but for any other value the amount must be included in the statement

When subtracting, we can use `dec` statement (short for *decrement*) in the same way:

```
dec x      rem *** subtract 1 from x ***
dec y, 3   rem *** subtract 3 from y ***
```

So why offer two ways to achieve the same thing? Using `inc` and `dec` allows the compiler to create more efficient code than is possible when using the started assignment approach.

The format for the `inc` statement is shown in FIG-3.10.

FIG-3.10

The inc Statement

`inc` `variable` [`,` `value`]

where:

variable is the variable whose value is to be incremented.

value is a numeric value giving the amount to be added to the variable. If *value* is omitted then 1 is added to the variable.

The format for the **dec** statement is given in FIG-3.11.

FIG-3.11

The dec Statement

dec **variable** [**,** **value**]

where:

variable is the variable whose value is to be decremented.

value is a numeric value giving the amount to be subtracted from the variable.

Operator Precedence

Of course, an arithmetic expression may have several parts to it as in the line

```
answer = no1 - 3 / v# * 2
```

and how the final result of such lines is calculated is determined by **operator precedence**.

If we have a complex arithmetic expression such as

```
answer = 12 + 18 / 3^2 - 6
```

then there's a potential problem about what should be done first when calculating the value of the expression. Will we start by adding 12 and 18 or subtracting 6 from 2, raising 3 to the power 2, or even dividing 18 by 3?

In fact, calculations are done in a very specific order according to a fixed set of rules. The rules are that the power operation (**^**) is always done first. After that comes remainder, multiplication and division with addition and subtraction done last. The power operator (**^**) is said to have a **higher priority** than remainder, multiplication and division; they in turn having a higher priority than addition and subtraction. So, to calculate the result of the statement above the computer begins by performing the calculation **3^2** which leaves us with:

```
answer = 12 + 18 / 9 - 6
```

Next the division operation is performed (18/9) giving:

```
answer = 12 + 2 - 6
```

The remaining operators, + and -, because they have the same priority, are performed on a left-to-right basis, meaning that we next calculate 12+2 giving:

```
answer = 14 - 6
```

Finally, the last calculation (14 - 6) is performed leaving

```
answer = 8
```

and the value 8 is stored in the variable *answer*.

Activity 3.9

What is the result of the calculation **12 - 5 * 12 / 10 - 5** ?

Using Parentheses

If we need to change the order in which calculations within an expression are performed, we can use parentheses. Expressions in parentheses are always done first. Therefore, if we write

```
answer = (12 + 18) / 9 - 6
```

then 12+18 will be calculated first, leaving:

```
answer = 30 / 9 - 6
```

The next calculation is 30 / 9 :

```
answer = 3 - 6  
answer = -3
```

► Remember we are dividing two integers so we get an integer result: 3.

An arithmetic expression can contain many sets of parentheses. Normally, the computer calculates the value in the parentheses by starting with the left-most set.

Activity 3.10

Show the steps involved in calculating the result of the expression

```
8 * (6-2) / (3-1)
```

If sets of parentheses are placed inside one another (this is known as **nested parentheses**), then the contents of the inner-most set is calculated first. Hence, in the expression

```
12 / (3 * (10 - 6) + 4)
```

the calculations are performed as follows:

(10 - 6)	giving	12 / (3*4+4)
3 * 4	giving	12 / (12 + 4)
12 + 4	giving	12 / 16
12 / 16	giving	0

The order of precedence for all arithmetic operators is shown in FIG-3.12.

FIG-3.12

Operator Priority

► Operators of equal priority are performed on a left-to-right basis.

Operator	Description	Priority
()	parentheses	1
^	power	2
*	multiplication	3
/	division	3
mod	remainder	3
+	addition	4
-	subtraction	4

Activity 3.11

Assuming a program begins with the lines `no1 = 12`, `no2 = 3`, and `no3 = 5` what would be the value stored in `answer` as a result of the line

```
answer = no1 / (4 + no2 - 1) * 5 - no3 ^ 2
```

Variable Range

When first learning to program, a favourite pastime of the beginner is to see how large a number the computer can handle, so people write lines such as:

```
no1 = 123456789000
```

They are often disappointed when the program crashes at this point.

There is a limit to the value that can be stored in a variable. That limit is determined by how much memory is allocated to a variable, and that differs from language to language.

Integer values in AGK BASIC can be in the range -2,147,483,648 to +2,147,483,647 while real values can be stored to about 7 decimal places.

String Operations

The + operator can also be used on string values to join them together. For example, if we write

```
a$ = "to" + "get"
```

then the value *toget* is stored in variable *a\$*. If we then continue with the line

```
b$ = a$ + "her"
```

b\$ will contain the value *together*, a result obtained by joining the contents of *a\$* to the string constant "her".

Activity 3.12

What value will be stored as a result of the statement

```
term$ = "abc"+"123"+"xyz"
```

The Print() Statement Again

We've already seen that the `Print()` command can be used to display values on the screen using lines such as:

```
Print(1)
Print("Hello")
```

We can also get the `Print()` statement to display the answer to a calculation. Hence,

```
Print(7+3)
```

will display the value 10 on the screen, while the statement

```
Print("Hello " + "again") rem ***Note the space after the o***
```

displays

```
Hello again
```

The `Print()` statement can also be used to display the value held within a variable. This means that if we follow the statement

```
number = 23
```

by the lines

```
Print(number)
Sync()
```

our program will display the value 23 on the screen, this being the value held in *number*. Real and string variables can be displayed in the same way. Hence the lines

```
name$ = "Charlotte"
weight# = 95.3
Print(name$)
Print(weight#)
Sync()
do
loop
```

will produce the output

```
Charlotte
95.3
```

Activity 3.13

A program contains the following lines of code:

```
number = 23
Print("number")
Print(number)
Sync()
```

What output will be produced by the two `Print()` statements?

Making Use of PrintC()

Although the `Print()` statement cannot display more than one value at a time, by using `PrintC()`, we can display two or more values on the same line of the screen.

For example, the code

```
capital$ = "Washington"
PrintC("The capital of the USA is ")
Print(capital$)
Sync()
do
loop
```

produces the following output on the screen:

```
The capital of the USA is Washington
```

►► The second output statement uses `Print()` in order to move the cursor to a new line after all output is complete.

Activity 3.14

Start a new project (called *Name*) that sets the contents of the variable *name\$* to *Jaqueline McKinnon* and then uses output statements that display the contents of *name\$* in such a way that the final message on the screen becomes:

Hello, Jaqueline McKinnon, how are you today?

Another way to output a sequence of strings, this time using only a single `Print()` statement, is to join those strings together so only one data value is being output:

```
Print("Hello, " + name$ + ", how are you today?")
```

Activity 3.15

Modify *Name* so that it uses a single `Print()` statement to perform all its output. Test and save the modified code.

Acquiring Data

Data input can come in many forms: mouse, joystick, screen press, and keyboard are perhaps the obvious ones. AGK allows all of these and we'll be looking at each of those methods later in the book.

Another way to retrieve information is to access the hardware's timer. AGK offers only two timer options. One gives you access to the time your program has been running to the nearest second, the other gives the same information but this time to the nearest one thousandth of a second.

Timer()

Many of the statements we have looked at so far require you to supply them with information. For example, you have to supply `Print()` with the information you want displayed, while `SetClearColor()` requires the strength of the red, green and blue components that make up the background colour you want to use. Values supplied to commands of this type are known as **in parameters**.

The `Timer()` statement, on the other hand, supplies you with information - the time your program has been running. When a command supplies you with a value, that value is known as a **return value**.

Syntax diagrams for commands that return a value have the format shown in FIG-3.13.

FIG-3.13

Statements that
Return a Value

return type Command Name () in parameters ()

Notice that *return type* is not enclosed. That is because the *return type* is information about the type of value returned by the command, but not part of how the command is written.

FIG-3.14

Timer()

The syntax diagram for the `Timer()` statement is shown in FIG-3.14.

float Timer ()

The diagram tells use that the `Timer()` statement returns a real value (also known as a **float** value) and that no *in parameters* are required by the statement. Notice that the parentheses must be included in the statement even though no information is placed within them. The actual value returned by `Timer()` is the time your program has been running to the nearest millisecond.

When a statement returns a value (as is the case with `Timer()`), generally we will want to do something with that returned value. Perhaps the most obvious thing to do is to store the result in a variable. Hence, we could add the line:

```
time_elapsed# = Timer()
```

We could then use that value in a calculation, for example

```
minutes = time_elapsed#/60
```

or simply display the value on the screen:

```
Print(time_elapsed#)
```

Activity 3.16

Start a new project called *Time*. Change the code in *main.agc* to read:

```
rem *** Get time passed ***
time_elapsed# = Timer()
do
    rem *** Display time ***
    PrintC("Time elapsed : ")
    Print(time_elapsed#)
    Sync()
loop
```

Compile and run the program.

You should see the time taken since the program started until the `Timer()` command was executed. This should be much less than 1 second.

Modify your program by moving the first two lines between the `do` and `loop` statements. Remember to change the indentation of the moved lines.

Compile and run the program. How does the output differ from the first version of the program?

The value returned by a statement doesn't have to be assigned to a variable. In the last exercise we assigned the value returned by `Timer()` to a variable then displayed the contents of that variable on the screen, but we can bypass the need for the variable by just printing the returned value directly with the line

```
Print(Timer())
```

which executes the `Timer()` statement then displays the value returned.

Activity 3.17

Modify *Time* so that the variable `time_elapsed#` is not required.

Test your modified program.

About Sync()

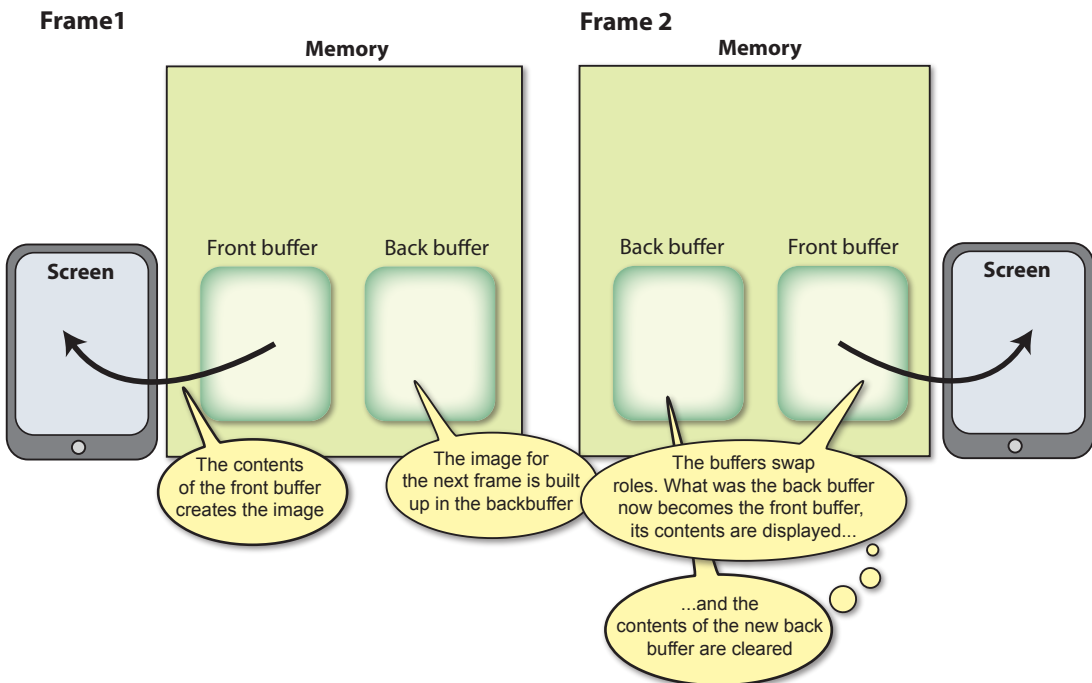
Let's take a moment out to get a deeper understanding of how `Sync()` works.

The contents of your screen are updated several times a second. Each update redraws the entire contents of the screen. Each redrawing is known as a **frame**.

To create a screen display, AGK reserves two areas of memory within your device. These areas of memory are known as **screen buffers**. The contents of one buffer are used to create the frame currently being displayed on the device's screen. This is known as the **screen buffer** or **front buffer**. At the same time, the contents of the second buffer (known as the **back buffer**) are being updated to contain the layout of the next frame.

FIG-3.15 shows how these buffers are used in the construction of a frame.

FIG-3.15 How the Screen Display is Produced



When a `Print()` or `PrintC()` statement is executed, the text to be displayed is copied into the current back buffer.

When a `Sync()` statement is executed, the two areas of memory swap function: what was the back buffer, becomes the front buffer and its contents appears on the screen; and what was the front buffer becomes the back buffer and its contents are cleared. It should be noted that handling the video buffers is not the `Sync()` statements only purpose, it also updates various other aspects of an application. We will examine these other aspects of `Sync()` in later chapters.

Understanding this will give you some insight as to where `Print()` and `PrintC()` statements need to be positioned within your program. Let's see how moving one of those statements affects the display of the *Time* project.

Activity 3.18

Since the message *Time elapsed* : never changes, try moving it before the `do` statement, then re-run your program.

What difference does this make to what is displayed?

After performing this, test, return the `PrintC()` statement to its original position after the `do` statement.

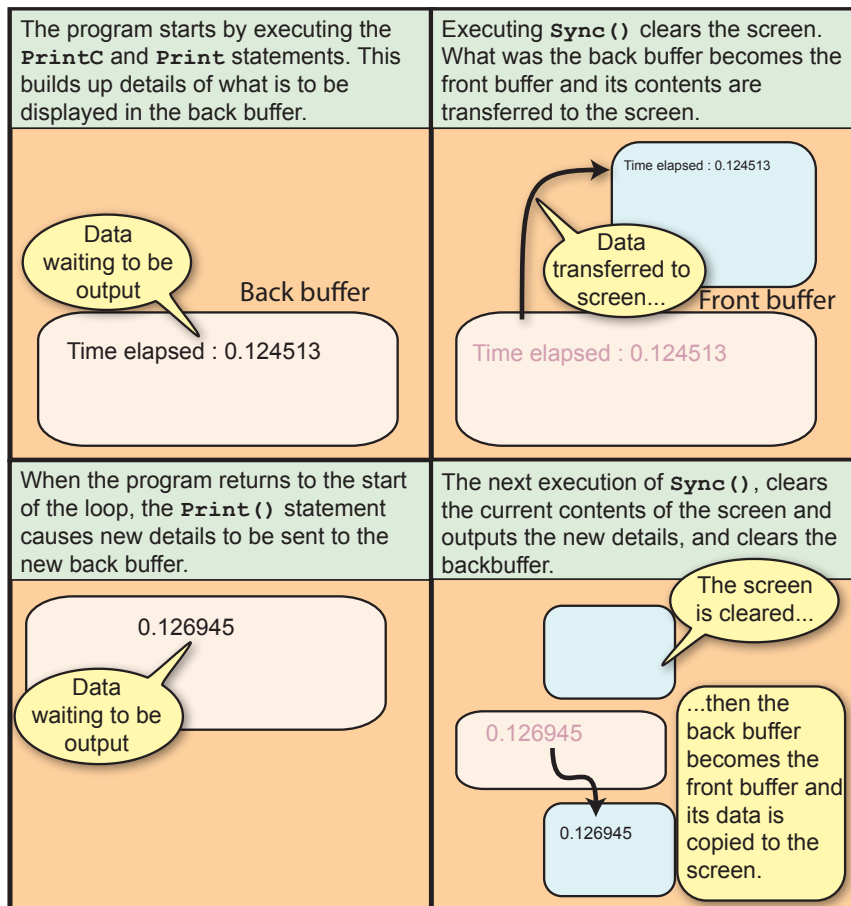
There is no need to resave your program.

So, why does the message no longer appear when we move it before the `do` statement? In fact, the message does appear, but it is gone so quickly that you won't have time to see it. After that, only the time appears.

FIG-3.15 explains the process involved when the first `PrintC()` statement appears before the `do`.

FIG-3.15

How Sync() Operates



The overall effect is that only values printed between one execution of `Sync()` and the next execution of `Sync()` will appear on the screen. If you want text to stay on the screen you need to reprint it between each execution of `Sync()`.

Timing Again

Most people are happier seeing a short period of time displayed in minutes and seconds rather than just seconds. To achieve this we can start by rounding the time elapsed to the nearest second using the line

➡ Remember, moving a real value to an integer variable causes that value to be rounded to the nearest integer.

```
total_seconds = Timer()
```

The number of minutes elapsed can now be calculated as *total_seconds* divided by 60:

```
minutes = total_seconds / 60
```

The remaining seconds (those not converted to minutes) give us the seconds part of our time. This is calculated as

```
seconds = total_seconds mod 60
```

➡ Remember, `mod` gives you the integer remainder after division has taken place.

The final version of our program is shown in FIG-3.16.

FIG-3.16

Displaying Time Elapsed in Minutes and Seconds

```
rem *** Display time elapsed in minutes and seconds ***

do
    rem *** Get time elapsed to nearest second ***
    total_seconds = Timer()
    rem *** Convert to minutes and seconds ***
    minutes = total_seconds / 60
    seconds = total_seconds mod 60
    rem *** Display the result ***
    PrintC("Time elapsed : ")
    PrintC(minutes)
    PrintC(":")
    Print(seconds)
    Sync()
loop
```

Activity 3.19

Modify your *Time* program to match the code given in FIG-3.16.

Compile and test your code.

ResetTimer()

Although the timer automatically starts tracking time from the moment your program begins executing, you can reset that timer to zero using the `ResetTimer()` statement (see FIG-3.17).

FIG-3.17

GetSeconds()

ResetTimer ()

Notice that this statement has neither in parameters nor a return value, instead it modifies the contents of a variable maintained by AGK itself.

GetMilliseconds()

While `Timer()` returns the time elapsed since the start of the program (or since the last execution of `ResetTimer()`) in seconds, you can have that same value in

milliseconds by using the `GetMilliseconds()` statement (see FIG-3.18).

FIG-3.18

`GetSeconds()`

integer `GetMilliseconds()`

GetSeconds()

If you are only interested in the time elapsed to the nearest second, you can use the `GetSeconds()` statement rather than `Timer().GetSeconds()`. `GetSeconds()` has the format shown in FIG-3.19.

FIG-3.19

`GetSeconds()`

integer `GetSeconds()`

Activity 3.20

Modify *Time* to use `GetSeconds()` instead of `Timer().GetSeconds()`. Test your new code.

Sleep()

It is possible to get a program to do nothing for a set period of time. As a general rule this is undesirable in a highly animated, interactive game, but for simple games such as those we will create in the early chapters of this book, getting a program to stop or slow down can be of use to us. For example, it may be used to give us the time to read a message on the screen.

Halting a program for a specific time is achieved using the `Sleep()` statement (see FIG-3.20).

FIG-3.20

`Sleep()`

`Sleep(imillisecs)`

where:

imillisecs is an integer value giving the time in milliseconds for which the program execution is to halt.

Activity 3.21

Modify your *Time* program adding the line

```
Sleep(2000)           rem *** halt for 2 seconds ***
```

immediately after the line containing `do`.

Run the program. How has the new line affected the program?

Generating Random Numbers

Often in a game we need to throw a dice, choose a card or think of a number. All of these are random events. That is to say, we cannot predict what value will be thrown on the dice, what card will be chosen, or what number some other person will think of.

To help emulate these type of situations AGK BASIC offers several statements for the generation and manipulation of random values.

Random()

The `Random()` statement is used to generate a random number between lower and upper limits (see FIG-3.21).

FIG-3.21

Random()

integer `Random` ((`ifrom` , `ito`))

where

ifrom is an integer giving the lowest value allowed.

ito is an integer giving the highest value allowed.

Activity 3.22

Start a new project (*Dice*) and create code to perform the following logic:

Throw a six-sided dice
Display the value thrown

Test the program by running it several times.

Save and close the project. We will return to this project frequently through the next few chapters.

The statement returns a random integer value in the range *ifrom* to *ito*. For example, if we wanted to simulate the throw of a dice, we could write

```
dice_throw = Random(1,6)
```

which would store a random value between 1 and 6 in *dice_throw*.

Notice that the syntax diagram tells us the parameters may be omitted allowing us to write a line such as

```
value = Random()
```

When no range of values is supplied, as in this example, the statement creates a random number in the range 0 to 65,535.

The program in FIG-3.22 shows another use of the `Random()` statement to create a random background colour for the app window.

FIG-3.22

Random Background
Colour

```
rem *** Cycle through random background colours ***
do
    rem *** Generate a random value for each colour ***
    red = Random(0,255)
    green = Random(0,255)
    blue = Random(0,255)

    rem *** Clear the screen using the new colour ***
    SetClearColor(red,green,blue)
    Sync()
loop
```

Activity 3.23

Start a new project (*Background*) and enter the code given in FIG-3.22.

What happens when you run the program?

Immediately after the `Sync()` statement, add the lines

```
rem *** wait for 0.5 seconds ***  
Sleep(500)
```

which will get the program to pause for half a second after each screen update. What difference does this make to the program?

Activity 3.24

Modify your *Background* project eliminating the need for the *red*, *green* and *blue* variables. Test your program to ensure it still works correctly.

We have already seen that the value returned by a statement can be assigned to a variable or displayed using a `Print()` statement, but we can also use the value returned by one statement as the parameter to another directly, without using a variable. Hence, we can replace the lines

```
red = Random(0,255)  
green = Random(0,255)  
blue = Random(0,255)  
SetClearColor(red,green,blue)
```

with the line

```
SetClearColor(Random(0,255),Random(0,255),Random(0,255))
```

SetRandomSeed()

Computers can't really think of a random number all by themselves. Actually, they cheat and use a mathematical algorithm to calculate an apparently random number. As long as you don't know that algorithm, you won't be able to predict what number the computer is going to come up with, but because the numbers generated are not truly random, they are often referred to as **pseudo random numbers**.

The mathematical formula used needs to be supplied with an initial number to get started. This is known as the **seed value**. This seed value determines exactly what set of pseudo random numbers will be generated - use the same seed value on a second occasion and exactly the same set of numbers will be generated. To prevent this happening, the random number generator in AGK defaults to using the time from the system clock as a seed value. This ensures that a different value is used each time a program is run.

If you want to use your own seed value, you can do so using the `SetRandomSeed()` statement. The most likely reason for doing this is to ensure you use the same seed value on each run and hence the same set of random values. Normally, of course, you wouldn't want the same set of values, but it can be extremely useful when trying to find mistakes in a program. The `SetRandomSeed()` has the syntax shown in FIG-3.23.

FIG-3.23

SetRandomSeed()

SetRandomSeed ((iseed))

where:

iseed is an integer value which is used as the start-up for the formula used in the generation of pseudo random values.

Activity 3.25

Modify your *Dice* project so that the program starts by setting the seed value to 12.

Run the program three times and check that the same number is generated each time. Remove the `SetRandomSeed()` line after testing is complete.

RandomSign()

A final statement that makes use of a random value is `RandomSign()` (see FIG-3.24).

FIG-3.24

RandomSign()

integer RandomSign ((ivalue))

where:

ivalue is an integer value which will be returned as either its original value or as a negated form of the original. In other words, if *ivalue* was 12 then the returned value will be either 12 or -12. Each return option has a 50% chance of occurring.

One possible use for such a statement is to emulate any situation with two possible outcomes each with an equal possibility of occurring - for example, the flip of a coin.

User Input

For many games, the most important method of obtaining data is from the user. The game player, will be moving a mouse, a joystick, tapping on the screen, or typing at the keyboard. AGK has statements available for handling all of these (and more) but at this stage using these statements are a bit beyond what we have learned. On the other hand, being able to enter simple values is very useful when trying to demonstrate some of the fundamental concepts in programming.

To allow us a simple way to enter integer values, two functions are included in the download for this book. These functions are:

The term **function** may, for the moment, be taken to have the same meaning as **program statement**.

SetUpButtons() This function sets up 12 round buttons on the right of the app window. The buttons are labelled 0 to 9, **←**(*backspace*) and **↵**(*Enter*).

GetButtonEntry() This function allows you to type in an integer value using the 12 buttons. Pressing the *backspace* button will remove the last character entered. Pressing *Enter* completes the data entry and returns the value entered.

The screen displayed when the buttons are used is shown in FIG-3.25.

FIG-3.25

Buttons Layout



The buttons are placed along the right edge to make them easy to press when the app is being used on a handheld device. If you want to use these new functions in any of your projects, you have to follow a few simple steps. These are shown in FIG-3.26.

FIG-3.26

Using the Buttons

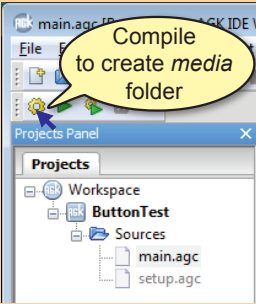
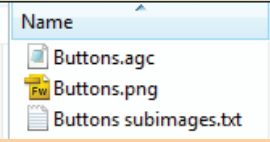
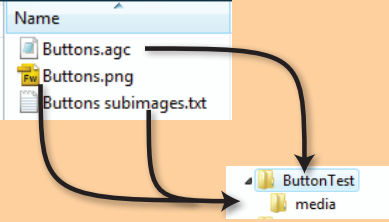
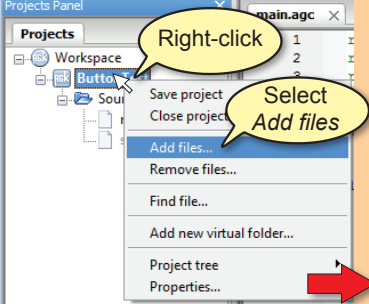
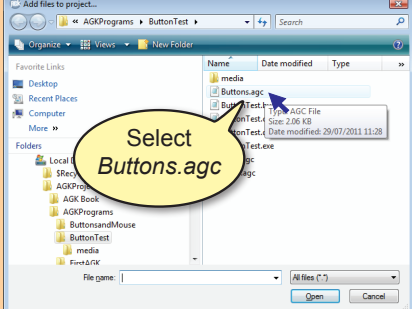
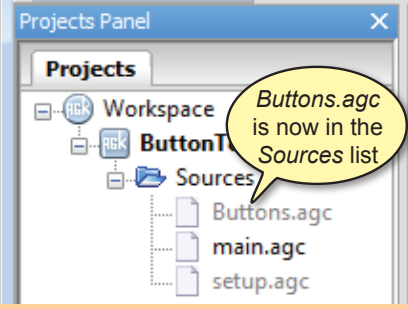
<p>We start by creating a new project (<i>ButtonTest</i>) in which to test the button routines. Compiling the default code creates a <i>media</i> subfolder.</p>	<p>The ZIP file download for Hands On AGK contains a folder called <i>Chapter3</i>. This folder contains 3 files.</p>
	<p>Files in <i>Chapter 3</i> folder</p> 
<p>The PNG and TXT files are copied to the project's <i>media</i> folder. The AGC file is copied to the project's main folder.</p>	<p>In the Projects Panel, right-click on <i>ButtonTest</i> and select Add files from the pop-up menu.</p>
	

FIG-3.26

Using the Buttons

<p>Double-click on the <i>Buttons.agc</i> file...</p>	<p>...to add the selected file to the Sources list in the Projects Panel.</p>
	
<p>In <i>main.agc</i>, we need to add the line #include "Buttons.agc" to allow the two functions held there to be used.</p>	<p>Now we can use SetUpButtons() to display the 12 buttons and GetButtonEntry() to accept input. The value is then displayed.</p>
<p>#include "Buttons.agc"</p>	<pre>#include "Buttons.agc" SetUpButtons() value_entered = GetButtonEntry() do PrintC("You entered ") Print(value_entered) Sync() loop</pre>

The complete code (with comments) for *main.agc* is shown in FIG-3.27.

FIG-3.27

Button Input

```
rem *** Command to include other source files used ***

#include "Buttons.agc"

rem *** Display the buttons ***
SetUpButtons()
rem *** Get an integer value from the buttons ***
value_entered = GetButtonEntry()
do
    rem *** Display the value entered ***
    PrintC("You entered ")
    Print(value_entered)
    Sync()
loop
```

The buttons are best suited to an app window optimised for the iPad's resolution of 1024 pixels high by 768 pixels wide, so we need to change the appropriate lines within the project's *setup.agc* to:

```
width=768
height=1024
```

Activity 3.26

Start a new project called *TestButtons*.

Compile the project in order to create the *media* subfolder.

From the *Chapter 3* folder of the files you downloaded for **Hands On AGK**, copy *Buttons.png* and *Buttons subtext.txt* into the *TestButtons* project's *media* folder.

From the *Chapter 3* folder copy *Buttons.agc* into the project's main folder.

Modify the contents of the project's *main.agc* so that the code matches that given in FIG-3.25.

Modify *setup.agc* so that the width is set to 768 and the height to 1024.

Compile and run the program, checking that you can enter and delete characters using the buttons.

Check that the number displayed when you press the *Enter* key matches the value you typed in.

Save and close your project.

Activity 3.27

Reload your *Dice* program.

Make the necessary adjusts to allow you to use button input in the program.

Modify the logic of *main.agc* to match the following structured English description:

- Display the set of input buttons
- Generate a random number between 0 and 9
- Display "Guess what my number is"
- Get a value entered on the buttons
- Display "My number was " and the game's number
- Display "Your guess was " and the value entered

The last two displays should appear on screen at the same time.

Compile and check your program by running it three times.

Resave your project.

We will be making use of the button input code in a few programs. The process for using the code is always the same:

- Copy the three files to the project's folders
- Add a **#include** statement to the start of *main.agc*
- Call the functions as required by the program logic
- Modify the dimensions specified in *setup.agc*

Summary

- The assignment statement takes the form

`variable = value`

value can be a constant, other variable, or an expression.

- The value assigned should be of the same type as the receiving variable.
- Arithmetic expressions can use the following operators:

`^ mod * / + -`

- Calculations are performed on the basis of highest priority operator first and a left-to-right basis.
- The power operator has the highest priority; multiplication and division and the remainder operator the next highest, followed by addition and subtraction.
- Terms enclosed in parentheses are always performed first.
- The + operator can be used to join strings.
- AGK uses a pseudo random number algorithm to create apparently random numbers within a specified range.
- The values generated are determined by an initial seed value.
- The default seed value for the algorithm is taken from the system's time.
- The seed value can also be set in the program code.
- Random integer values within a specified range can be created.

Testing Sequential Code

The programs in this chapter are very simple ones, with the statements being executed one after the other, starting with the first and ending with the last. In other words, the programs are sequential in structure.

Every program we write needs to be tested. For a simple sequential program which accepts user input, the minimum testing involves thinking of a value to be entered, predicting what result this value should produce, and then running the program to check that we do indeed obtain the expected result from that test data.

The program below (see FIG-3.28) reads in a value from the buttons and displays the square root of that number.

FIG-3.28

Calculating the
Square Root

```
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Display prompt ***
Print("Enter a number : ")
Sync()
Sleep(2000)
rem *** Get value ***
no = GetButtonEntry()
rem *** Calculate square root ***
sqroot# = no^0.5
do
    rem *** Display result ***
    PrintC("Square root of ")
    PrintC(no)
    PrintC(" is ")
    Print(sqroot#)
    Sync()
loop
```

Activity 3.28

Start a new project called *SquareRoot*.

Perform the operations necessary so you can make use of button input in the program. Set the app windows dimensions to 1024 x 768.

Recode *main.agc* to match the lines given in FIG-3.28.

Compile the program but do not run it.

To test this program we might decide to enter the value 16 with the expectation of the displayed result being 4.

Activity 3.29

Test *SquareRoot* using the value 16.

Did you achieve the expected result?

Perhaps that one test would seem sufficient to say that the program is functioning correctly. However, a more cautious person might try a few more values just to make sure. But what values should be chosen? Should we try 25 or 9, 3 or 7?

As a general rule it is best to think carefully about what values you choose as test data. A few carefully chosen values may show up problems when many more randomly chosen values show nothing.

When the test data involves numeric values only, perhaps the most obvious categories are positive numbers, negative numbers and zero (which is neither negative or positive).

We have already tried a positive number (16), so perhaps we should try -9, say, and, of course, zero.

But in each case it is important that you work out the expected result before entering your test data into the program - otherwise you have no way of knowing if the results you are seeing on the screen are correct.

Activity 3.30

What results would you expect from *SquareRoot* if your test data was
0 and -9

Run the program with these test values and check that the expected results are produced.

When the value being entered by the user is a string, perhaps the test data could be:

- a string with zero characters (just press the *Enter* when asked for data)
- a string with only a single character
- a string containing multiple characters

Of course, these suggestions for creating test data will almost certainly need to be modified depending on the nature of the program you are testing.

Solutions

Activity 3.1

- | | | | |
|------------|------------|------------|-----------|
| a) Integer | b) String | c) Integer | d) Real |
| e) String | f) Integer | g) Real | h) String |
| i) String | j) Real | | |

Activity 3.2

- | | |
|--------------|------------------|
| a) -12 | integer constant |
| b) Elizabeth | string constant |
| c) 3.14 | real constant |
| d) 27.0 | real constant |

Activity 3.3

- a) Valid
- b) Invalid. Stores 13
- c) Invalid - not a string variable
- d) Invalid - remove \$ from variable name or put double quotes round the 5.
- e) Invalid. Must be double quotes, not single quotes.
- f) Valid.

Activity 3.4

- a) Valid
- b) Invalid. Must start with a letter
- c) Invalid. Names cannot be within quotes.
- d) Valid
- e) Invalid. Spaces are not allowed in a name
- f) Invalid. # must appear at the end of the name
- g) Invalid, then is a BASIC keyword
- h) Valid

Activity 3.5

- a) `desc$="tall"`
- b) `result#= 12.34`

Activity 3.6

- a) Valid
- b) Invalid. Fraction part rounded
- c) Invalid. A string cannot be copied to an integer variable
- d) Valid
- e) Invalid. A real cannot be copied to a string variable
- f) Invalid. A string cannot be copied to a real variable

Activity 3.7

- a) 2
- b) -1
- c) 5
- d) -4

Activity 3.8

- a) `no2` is 16
- b) `x#` is 82.18
- c) `no3` is zero
- d) `no4` is 9
- e) `m#` is 0.0
- f) `v2#` is 40.99
- g) `no1` is 3
- h) `no5` is -2

Activity 3.9

The result is 1

The expression is calculated as follows:

```
12-5* 12/10-5
12-60/10-5
12-6-5
6-5
1
```

In fact, AGK BASIC doesn't currently abide by the rules of priority completely with it performing the division before the multiplication in this example which results in an answer of 2 rather than 1!

Activity 3.10

Steps:

```
8*(6-2)/(3-1)
8*4/(3-1)
8*4/2
32/2
16
```

Activity 3.11

```
answer = no1 / (4 + no2 - 1) * 5 - no3 ^ 2
answer = 12 / (4 + 3 - 1) * 5 - 5 ^ 2
answer = 12 / (7 - 1) * 5 - 5 ^ 2
answer = 12 / 6 * 5 - 5 ^ 2
answer = 2 * 5 - 25
answer = 10 - 25
answer = -15
```

Activity 3.12

`term$` will hold the string `abc123xyz`

Activity 3.13

Output:
number
23

Activity 3.14

The program code:

```
name$ = "Jaqueline McKinnon"
do
  PrintC("Hello, ")
  PrintC(name$)
  Print(", how are you today?")
  Sync()
loop
```

Note the spaces inside the quotes to make sure there are gaps either side of the name.

Activity 3.15

The program code:

```
name$ = "Jaqueline McKinnon"
do
  Print("Hello, "+name$+" , how are you today?")
  Sync()
loop
```

Activity 3.16

Modified code:

```
do
  rem *** Get time passed ***
```

```

time_elapsed# = Timer()
rem *** Display time ***
PrintC("Time elapsed : ")
Print(time_elapsed#)
Sync()
loop

```

The time displayed on the screen now updates continuously.

Activity 3.17

Modified code:

```

do
rem *** Display time passed ***
PrintC("Time elapsed : ")
Print(Timer())
Sync()
loop

```

Activity 3.18

Modified code:

```

PrintC("Time elapsed : ")
do
rem *** Display time passed ***
Print(Timer())
Sync()
loop

```

Each time the `Sync()` statement is executed, only the contents of `Print()` or `PrintC()` statements executed since the previous execution of `Sync()` are displayed. Since the `PrintC()` statement above is executed only once, its message disappears the second time the `Sync()` statement is executed.

Activity 3.19

No solution required.

Activity 3.20

Modified code:

```

rem *** Display time elapsed in ***
rem *** minutes and seconds ***
do
rem *** Get time elapsed to nearest second ***
total_seconds = GetSeconds()
rem *** Convert to minutes and seconds ***
minutes = total_seconds / 60
seconds = total_seconds mod 60
rem *** Display the result ***
PrintC("Time elapsed : ")
PrintC(minutes)
PrintC(":")
PrintC(seconds)
Sync()
loop

```

Activity 3.21

Modified code:

```

rem *** Display time elapsed in ***
rem *** minutes and seconds ***
do
Sleep(2000) rem *** halt for 2 seconds ***
rem *** Get time elapsed to nearest second ***
total_seconds = GetSeconds()
rem *** Convert to minutes and seconds ***
minutes = total_seconds / 60
seconds = total_seconds mod 60
rem *** Display the result ***
PrintC("Time elapsed : ")
PrintC(minutes)
PrintC(":")
PrintC(seconds)
Sync()
loop

```

The change means that the screen is only updated every 2 seconds so we see the time pass in 2 second steps.

Activity 3.22

Program code:

```

rem *** Dice program ***
rem *** Simulates the roll of a 6-sided dice ***
rem *** Throw dice ***
dice = Random(1,6)
do
rem *** Display value thrown ***
PrintC("Value thrown was : ")
Print(dice)
Sync()
loop

```

Activity 3.23

The colours change so quickly that there is not time to update the whole background before the colour changes again so bands of colour appear.

Modified code:

```

rem *** Cycle through random background colours ***
do
rem *** Generate value for each colour ***
red = Random(0,255)
green = Random(0,255)
blue = Random(0,255)

rem Clear the screen using the new colour ***
SetClearColor(red,green,blue)
Sync()
rem *** wait for 0.5 seconds ***
Sleep(500)
loop

```

Now there is enough time to show the selected colour over the whole background before another colour is generated.

Activity 3.24

Modified Code:

```

rem *** Cycle through random background colours ***
do
rem Clear the screen using random colour ***
SetClearColor(Random(0,255),Random(0,255),
⌘Random(0,255))
Sync()
rem *** wait for 0.5 seconds ***
Sleep(500)
loop

```

Note The symbol `⌘` is used to indicate the continuation of a single line of code.

Activity 3.25

Modified code:

```

rem *** Dice program ***
rem *** Simulates the roll of a 6-sided dice ***

rem *** Seed random number generator ***
SetRandomSeed(12)
rem *** Throw dice ***
dice = Random(1,6)
do
rem *** Display value thrown ***
PrintC("Value thrown was : ")
Print(dice)
Sync()
loop

```

The program always generates a 6.

Activity 3.26

No solution required.

Activity 3.27

Reload your *Dice* project.

Modify the *startup.agc* file setting the width to 768 and the height to 1024.

From the *Chapter 3* folder of the files you downloaded for **Hands On AGK**, copy *Buttons.png* and *Buttons subtext.txt* into the project's *media* folder.

From the *Chapter 3* folder copy *Buttons.agc* into the project's main folder.

Right click on **Dice** in the Projects Panel.

Select **Add files** from the popup menu.

Select *Buttons.agc* from the files listed.

Program code:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
do
    rem *** Display values ***
    PrintC("My number was : ")
    Print(dice)
    PrintC("Your guess was : ")
    Print(guess)
    Sync()
loop
```

Activity 3.28

Start a new project called *SquareRoot*.

Compile the project to create the media folder.

Modify the *startup.agc* file setting the width to 768 and the height to 1024.

From the *Chapter 3* folder of the files you downloaded for **Hands On AGK**, copy *Buttons.png* and *Buttons subtext.txt* into the project's *media* folder.

From the *Chapter 3* folder copy *Buttons.agc* into the project's main folder.

Right click on **SquareRoot** in the Projects Panel.

Select **Add files** from the popup menu.

Select *Buttons.agc* from the files listed.

Change the contents of *main.agc* to match that given in FIG-3.24.

Compile the program.

Activity 3.29

Running the program using the value of 16 gives the result 4.0.

Activity 3.30

The expected result using the value zero would be zero.

Using -9 should result in an error since negative values do not have a square root.

4

Selection

In this Chapter:

- ☐ **if..endif** Statement
- ☐ Conditions
- ☐ Relational Operators
- ☐ Boolean Operators
- ☐ **if..then** Statement
- ☐ Nested if Statements
- ☐ Testing Selection Structures

Binary Selection

Introduction

As we saw in structured English, many algorithms need to perform an action only when a specified condition is met. The general form for this statement was:

```
IF condition THEN
    action
ENDIF
```

Hence, in our guessing game, we described the response to a correct guess as:

```
IF guess = dice THEN
    Say "Correct"
ENDIF
```

As we'll see, AGK BASIC also makes use of an **if** statement to handle such situations.

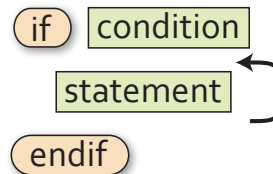
if

In its simplest form, the **if** statement in AGK BASIC takes the format shown in FIG-4.1.

FIG-4.1

if (format 1)

► Unlike the IF in structured English, AGK BASIC does not use the word **then**.



where:

condition is any term which can be reduced to a true or false value.

statement is any executable AGK BASIC statement.

The diagram also tells us that we can have as many statements between *condition* and **endif** as we require.

If *condition* evaluates to true, then the set of statements between the **if** and **endif** terms are executed; if *condition* evaluates to false, then the set of statements are ignored and execution moves on to any statements following the **endif** term.

Condition

Generally, the condition will be an expression in which the relationship between two quantities is compared. For example, the condition

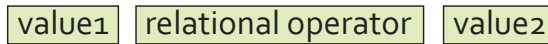
```
no < 0
```

will be true if the content of the variable *no* is less than zero (i.e. negative).

A condition is sometimes referred to as a **Boolean expression** and has the general format given in FIG-4.2.

FIG-4.2

Boolean
Expression



where:

value1 and **value2**

may be constants, variables, or expressions.

relational operator

is one of the symbols given in FIG-4.3.

FIG-4.3

The Relational
Operators

English	Symbol
is less than	<
is less than or equal to	<=
is greater than	>
is greater than or equal to	>=
is equal to	=
is not equal to	<>

From our syntax diagram, we can see that each of the following are valid conditions:

```
no1 < 7
answer# <> no1# * 2
gender$ = "female"
```

The values being compared should normally be of the same type, but it is acceptable to mix integer and real numeric values as in the conditions:

```
v > x#
t# < 12
```

However, it is not possible to compare a numeric against a string value. Therefore, conditions such as

```
name$ = 34
no1 <> "16"
```

are invalid.

Activity 4.1

Which of the following are NOT valid Boolean expressions?

- a) `no1 < 0` b) `name$ = "Fred"` c) `no1 * 3 >= no2 - 6`
d) `v# => 12.0` e) `total <> "0"` f) `address$ = 14 High Street`

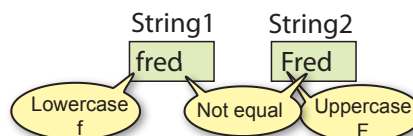
When two strings are checked for equality as in the condition

```
if name$ = "Fred"
```

the condition will only be considered true if the match is an exact one. Even the slightest difference between the two strings will return a *false* result (see FIG-4.4).

FIG-4.4

String
Comparison 1



Spaces count as characters too. So if one or more spaces are included in a string, their number and positions within two strings must also match if the strings are to be considered equal. Since spaces are so important, you will occasionally see the space represented within a string as a triangle. So rather than show the contents of a string as

```
Hello world
```

you may see

```
Hello△world
```

This is only done when clarification of the exact contents of a string is required. For example, the strings *hello* and *hello△* are not equal because the second string contains a space character after the letter *o*.

Not only is it valid to test if two string values are equal, or not, as in the conditions

```
if name$ = "Fred"  
if village$ <> "Turok"
```

it is also valid to test if one string value is greater or less than another. For example, it is true that

```
"B" > "A"
```

Such a condition is considered true not because B comes after A in the alphabet, but because the coding used within the computer to store a "B" has a greater numeric value than the code used to store "A".

The method of coding characters is known as ASCII (American Standard Code for Information Interchange). This coding system is given in Appendix A at the back of the book.

If you are comparing strings which only contain letters, then one string is less than another if that first string would appear first in an alphabetically ordered list. Hence,

```
"Aardvark"    is less than    "Abolish"
```

But watch out for upper and lower case letters. All upper case letters are less than all lower case letters. Hence, the condition

```
"A" < "a"
```

is true.

If two strings differ in length, with the shorter matching the first part of the longer as

```
"abc" < "abcd"
```

then the shorter string is considered to be less than the longer string. Also, because the computer compares strings using their internal codes, it can make sense of a condition such as

```
"$" < "?"
```

which is also considered true since the \$ sign has a smaller value than the ? character

in the ASCII coding system.

Activity 4.2

Determine the result of each of the following conditions (true or false). You may have to examine the ASCII coding at the end of the book for f).

- a) `"wxy" = "w xy"` b) `"def" < "defg"` c) `"AB" < "BA"`
d) `"cat" = "cat."` e) `"dog" = "Dog"` f) `"*" > "&"`

Structured English to Code

It is not always obvious how to translate an IF statement written in structured English. In fact, some may take a great deal of coding. For example, the structured English

```
IF the text entered contains any punctuation marks THEN
    Remove the punctuation marks from the text
ENDIF
```

would require several lines of programming code to achieve. On the other hand, some statements that might look difficult to code are very simple:

Structured English:

```
IF number is negative THEN
    Make it positive
ENDIF
```

Code:

```
if number < 0
    number = -number
endif
```

Structured English:

```
IF number is even THEN
    Display "Even number"
ENDIF
```

Code:

```
if number mod 2 = 0
    Print("Even number")
endif
```

► Notice the use of indentation in the program listings. BASIC does not demand that this be done, but indentation makes a program easier to read - this is particularly true when more complex programs are written.

If you wanted the display to update immediately, you would also add `sync()` after the `print()` statement.

Place the lines
do
loop
at the end of your
code.

Activity 4.3

Start a new project *EnglishToCode*. The program will accept values from the screen buttons we used previously. The program should implement the following logic:

```
Read in values for no1 and no2
IF no1 is exactly divisible by no2 THEN
    Display "Exactly divisible"
ENDIF
```

Test your program.

Using if

Activity 4.4

Load *Dice*, the project you created in Chapter 3.

Modify the program so that, after the player has typed in his guess, the program displays the word *Wrong* if the *guess* and *dice* values are not equal.

Test and save your program.

As we have already said, the syntax diagram for the `if` statement shows us that we can have more than one statement between the condition and the term `endif`. For example, if a game which used two dice required the dice to be re-thrown if they both showed the same value, then we would write:

```
if dice1 = dice2
    dice1 = Random(1,6)
    dice2 = Random(1,6)
endif
```

Activity 4.5

Modify the latest version of *Dice* so that, when the number generated differs from the guess, the program displays the word *Wrong* and also the difference between the two numbers. For example if the computer generates the value 8 and the player guesses 3 then the output would be:

```
Wrong. You were out by 5
My number was 8
Your guess was 3
```

Compound Conditions - the *and* and *or* Operators

Two or more simple conditions (like those given earlier) can be combined using either the term `and` or the term `or` (just as we did in structured English in Chapter 1).

The term `and` should be used when we need two conditions to be true before an action should be carried out. For example, if a game requires you to throw two sixes to win, this could be written as:

```
dice1 = Random(1,6)
dice2 = Random(1,6)
if dice1 = 6 and dice2 = 6
    Print("You win!")
    Sync()
endif
```

The statements `Print("You win!")` and `Sync()` will only be executed if both conditions, `dice1 = 6` and `dice2 = 6`, are true.

You may recall from Chapter 1 that there are four possible combinations for an `if` statement containing two simple expressions. Because these two conditions are linked by the `and` operator, the overall result will only be true when both conditions are true. These combinations are shown in FIG-4.5.

FIG-4.5

AND
Combinations

condition 1	condition 2	condition 1 AND condition 2
false	false	false
false	true	false
true	false	false
true	true	true

We link conditions using the `or` operator when we require only one of the conditions given to be true. For example, if a dice game produces a win when the total of two dice is either 7 or 11, we could write the code for this as:

```
dice1 = Random(1,6)
dice2 = Random(1,6)
total = dice1 + dice2
if total = 7 or total = 11
    Print("You win!")
    Sync()
endif
```

The four possible combinations for two conditions linked by an `or` are shown in FIG-4.6.

FIG-4.6

OR
Combinations

condition 1	condition 2	condition 1 OR condition 2
false	false	false
false	true	true
true	false	true
true	true	true

When you use multiple conditions linked with `and` or `or`, each condition must be properly formed; you cannot shorten things the way you might in standard English. Hence, the compiler would not accept

```
if total = 7 or 11
```

Activity 4.6

Start a new project called *TwoDice*. Create a program using the two-dice code given above.

Add statements to display the values thrown on the two dice. This should appear irrespective of the values thrown. You will have to reposition the `Sync()` statement to get the program to operate correctly.

Test and save your program.

There is no limit to the number of conditions that can be linked using `and` and `or`. For example, a statement of the form

```
IF condition1 AND condition2 AND condition3
```

means that all three conditions must be true, while the statement

```
IF condition1 OR condition2 OR condition3
```

means that at least one of the conditions must be true.

Activity 4.7

Modify your *TwoDice* project so that the *You win!* message also appears if both dice have equal values.

Test and save your program.

Activity 4.8

Start a new project called *ThreeDice*.

In this game three dice are thrown. If at least two dice show the same value, the player has won.

Write a program which implements the following logic:

```
Throw all three dice
IF any two dice match THEN
    Display "You win!"
ENDIF
Display the value of each dice
```

Test and save your program.

A complex condition can also contain a mix of **and** and **or** operators. An obvious example of this is the description of how to save a file in AGK:

```
IF Save button pressed OR Ctrl key down AND S key pressed THEN
    Save current file
ENDIF
```

The trouble with conditions like this is that they are open to more than one interpretation. We could take it to mean:

that we must press the *S* key while either clicking on the **Save** button or holding down the *Ctrl* key

rather than the intended

either clicking on the **Save** button or holding down the *Ctrl* key while pressing the *S* key.

Once we start to create conditions containing both **and** and **or** operators, we need to be aware that Boolean operators (AND, OR and NOT) have a priority order just as arithmetic operators do. In a condition that contains both **and** and **or**, the **and** operator takes precedence over the **or** operator. Knowing this eliminates any ambiguity in the conditions for saving a file in the example above.

The normal rule of performing the **and** operation before **or** can be modified by the use of parentheses. Expressions within parentheses are always evaluated first. Hence, if we really did have to click on the press the *S* key while pressing the **Save** button or holding down the *Ctrl* key, we would write the condition as

```
(Save button pressed OR Ctrl key down) AND S key pressed
```

Activity 4.9

Write down formal conditions (including any necessary parentheses) for the following situations:

- a) In the game of Monopoly any one of three situations causes your piece to “go to jail”. These are: landing on the “Go to Jail” square, picking up a “Go to Jail” card, and, throwing the same value on both dice three times in a row.
- b) In a video game, one way to win is to collect 10,000 gold pieces; an alternative is to free the princess from the tower and slay the dragon.
- c) In a game of cards, you lose 100 points if you hold either the King or Queen of Spades when the Ace of Diamonds is played.

The not Operator

AGK BASIC’s **not** operator works in exactly the same way as that described in Chapter 1. It is used to negate the final result of a Boolean expression.

In the *ThreeDice* project you created in Activity 4.8, the **if** statement used was

```
if dice1 = dice2 or dice1 = dice3 or dice2 = dice3
    Print("You win")
endif
```

Now, if we wanted to change the game to display “You lose” instead of “You win” then we would have to test for the opposite condition.

Activity 4.10

Without using the **not** operator, write down the condition that should be tested when displaying “You lose” in the dice game.

As you can see, working out the opposite condition takes a few moments - you may even have got it wrong on your first attempt. It’s much easier, given that you already have the condition required for the “You win” message, just to add a **not** to the condition:

```
if not(dice1 = dice2 or dice1 = dice3 or dice2 = dice3)
    Print("You lose")
endif
```

Note that the original condition is placed in parentheses. This is because the **not** operator has an even higher priority than **and** and **or**. Without the parenthesis, the **not** operation would be applied to the first term only - **dice1 = dice2**.

The Boolean operator priority is shown in FIG-4.7.

FIG-4.7

Boolean Priority

Operator	Priority
()	1
not	2
and	3
or	4

else - Creating Two Alternative Actions

In its present form the `if` statement allows us to perform an action when a given condition is met. But sometimes we need to perform an action only when the condition is not met. For example, when the user has to guess the number generated by the computer, we use an `if` statement to display the word “Correct” when the user guesses the number correctly:

```
if guess = number
    Print("Correct")
endif
```

However, shouldn't we display an alternative message when the player is wrong? One way to do this is to follow the first `if` statement with another testing the opposite condition:

```
if guess = dice
    Print("Correct")
endif

if not guess = dice
    Print("Wrong")
endif
```

We could also have written

```
if guess <> dice
```

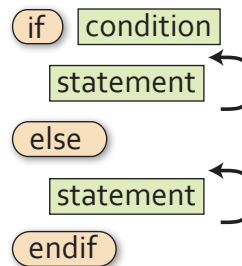
Although this will work, it's not very efficient since we always have to test both conditions - and the second condition can't be true if the first one is! As an alternative, we can add the word `else` to our original `if` statement and follow this by the action we wish to have carried out when the stated condition is false:

```
if guess = dice
    Print("Correct")
else
    Print("Wrong")
endif
```

This gives us the longer version of the `if` statement format as shown in FIG-4.8.

FIG-4.8

`if ..else..endif`



Note that we can have an unlimited number of statements between `else` and `endif`.

Activity 4.11

In your *Dice* program, modify the existing `if` statement to match the version given above so that either “Correct” or “Wrong” is displayed. Remove the code to calculate the difference between the *dice* and *guess* values.

Test and save your program.

Activity 4.12

Start a new project called *TwoNumbers*.

Make use of the button input files to read in two integer values and then display the smaller of the two numbers. Also display a message indicating whether this smaller value is an odd or even number.

The program should use the following logic:

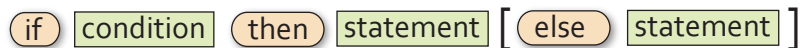
```
Display a prompt message for first number
Read the first number
Display a prompt message for the second number
Read the second number
IF first number is less than the second number THEN
    Set answer to first number
ELSE
    Set answer to second number
ENDIF
Display answer
IF answer is an even number THEN
    Display "Even"
ELSE
    Display "Odd"
ENDIF
```

The Other if Statement

AGK BASIC actually offers a second version of the `if` statement which has the format shown in FIG-4.9.

FIG-4.9

if..then..else



As with the previous `if` statement, the `else` section is optional but this version uses the word `then` and omits the `endif` term. Also, as the syntax diagram shows, you are restricted to a single statement after the `then` and `else` terms.

A major restriction when using this version of the `if` statement is that the `else` section of the statement must appear on the same line of the screen as the rest of the statement.

This means that the code you added in Activity 4.10 would have to be written as:

```
if dice = guess then Print("Correct") else Print("Wrong")
```

This lack of indented layout is enough to have the hardened programmer throw up her hands in horror!

Even when a single statement within the `if` statement is sufficient for the logic being coded, it is probably best to avoid this version of the `if` statement, since the requirement to place the `if` and `else` terms on the same line does not allow a good layout for the program code.

Activity 4.13

- a) What is a Boolean expression?
- b) How many relational operators are there?
- c) If a condition contains **and**, **or** and **not** operators, which will be performed first?

Summary

- Conditional statements are created using the **if** statement.
- A Boolean expression is one which gives a result of either true or false.
- Conditions linked by the **and** operator must all be true for the overall result to be true.
- Only one of the conditions linked by the **or** operator needs to be true for the overall result to be true.
- When the **not** operation is applied to a condition, it reverses the overall result.
- The statements following a condition are only executed if that condition is true.
- Statements following the term **else** are only executed if the condition is false.
- A second version of the **if** statement is available in AGK BASIC in which **if** and **else** must appear on the same line.

Multi-Way Selection

Introduction

A single `if` statement is fine if all we want to do is perform one of two alternative actions, but what if we need to perform one action from three or more possible actions? How can we create code to deal with such a situation?

In structured English we use a modified IF statement of the form:

```
IF
    condition 1:
        action1
    condition 2:
        action 2
ELSE
    action 3
ENDIF
```

However, this structure is not available in AGK BASIC and hence we must find some other way to implement multi-way selection.

Nested if Statements

There are two main ways of achieving multi-way selection in AGK BASIC. One is to use nested `if` statements - where one `if` statement is placed within another. For example, let's assume in the *Dice* project that we want to display one of three messages: *Correct*, *Your guess is too high*, or *Your guess is too low*. Our previous solution allowed for two alternative messages, *Correct* or *Wrong*, and was coded as:

```
if guess = dice
    Print("Correct")
else
    Print("Wrong")
endif
```

In this new problem the `Print("Wrong")` statement needs to be replaced by the two alternatives, *Your guess is too high* or *Your guess is too low*. But we already know how to deal with two alternatives - use an `if` statement. The `if` statement for this situation would be:

```
if guess > dice
    Print("Your guess is too high")
else
    Print("Your guess is too low")
endif
```

If we now remove the `Print ("Wrong")` statement from our earlier code and substitute the four lines given above, we get:

```
if guess = dice
    Print("Correct")
else
    if guess > dice
        Print("Your guess is too high")
    else
        Print("Your guess is too low")
    endif
endif
```

We have created a nested `if` situation, where the `if guess > dice` statement is inside the `else` section of the `if guess = dice` statement.

Activity 4.14

Modify your *Dice* project so that the game will respond with one of three messages as shown in the code given above.

Test and save your program.

Activity 4.15

Start a new project called *Number*.

The program should generate a random number in the range -12 to +12.

The program should now display one of the following messages: *Negative* (if the number is less than zero), *Zero* (if the number is zero), or *Positive* (if the number is greater than zero). Finally, the value of the number should also be displayed.

Test and save your program.

There is no limit to the number of `if` statements that can be nested. Hence, if we required four alternative actions, we might use three nested `if` statements, while four nested `if` statements could handle five alternative actions. To demonstrate this we'll take our number guessing game a stage further and have it display one of five possible messages:

<i>Your guess is too high</i>	(if the guess is more than 2 above the dice)
<i>Your guess is slightly too high</i>	(if the guess is no more than 2 above the dice)
<i>Correct</i>	(if the guess equals the dice)
<i>Your guess is slightly too low</i>	(if the guess is no more than 2 below the dice)
<i>Your guess is too low</i>	(if the guess is more than 2 below the dice)

Activity 4.16

Reload *Dice*.

Modify the code so that it displays one of the five messages given above under the appropriate conditions. (HINT: You'll have to calculate the difference between the *guess* and *dice* values again.)

Test and save your program.

➡ **Mutually exclusive conditions** refers to a set of conditions where no more than one of those conditions can be true at the same time.

When we have a set of mutually exclusive conditions, as in the *Dice* example given above, following the standard layout of indenting within an `if` statement results in the layout shown below:

```
if diff > 2
    Print("Your guess is too low")
else
    if diff > 0
        Print("Your guess is slightly too low")
    else
        if diff = 0
```

```

        Print("Correct")
    else
        if diff >= -2
            Print("Your guess is slightly too high")
        else
            Print("Your guess is too high")
        endif
    endif
endif
endif
endif

```

In a situation that included even more options, the indentation can be so extreme that you may reach the right-hand margin! To solve this problem we often re-arrange the layout of nested `if` statements to be

```

if diff > 2
    Print("Your guess is too low")
else if diff > 0
    Print("Your guess is slightly too low")
else if diff = 0
    Print("Correct")
else if diff >= -2
    Print("Your guess is slightly too high")
else
    Print("Your guess is too high")
endif endif endif endif

```

with each option given the same indentation as the last, and with the closing set of `endif` keywords placed on a single line. This gives a much neater layout which is still easy to follow.

Activity 4.17

Modify the layout of your *Dice* program to conform to this new layout style for multi-way selection. Resave your project.

elseif

The only problem with the previous solution is the need for so many `endif` terms at the end of the selection process. To avoid this we can replace the separate `else if` terms with the single word `elseif`. When we do this, only a single `endif` term is required at the end of the structure:

```

if diff > 2
    Print("Your guess is too low")
elseif diff > 0
    Print("Your guess is slightly too low")
elseif diff = 0
    Print("Correct")
elseif diff >= -2
    Print("Your guess is slightly too high")
else
    Print("Your guess is too high")
endif

```

Activity 4.18

Modify *Dice* to use the `elseif` term. Resave your project.

The select Statement

An alternative, and often clearer, way to deal with choosing one action from many is to employ the `select` statement. The simplest way to explain the operation of the `select` statement is simply to give you an example.

In the code snippet given below we display the name of the day of week corresponding to the number generated. For example, 1 results in the word *Sunday* being displayed.

```
day = Random(0,8)
select day
  case 1:
    Print("Sunday")
  endcase
  case 2:
    Print("Monday")
  endcase
  case 3:
    Print("Tuesday")
  endcase
  case 4:
    Print("Wednesday")
  endcase
  case 5:
    Print("Thursday")
  endcase
  case 6:
    Print("Friday")
  endcase
  case 7:
    Print("Saturday")
  endcase
endselect
Print(day)
Sync()
```

Once a value for `day` has been generated, the `select` statement chooses the `case` statement that matches that value and executes the code given within that section. All other `case` statements are ignored and any instructions following the `endselect` statement are executed. For example, if `day = 3`, then the statement given beside `case 3` will be executed (i.e. `Print("Tuesday")`) and the remainder of the whole `select...endselect` structure ignored with the next statement executed being `Print(day)`. If `day` were to be assigned a value not given in any of the `case` statements (e.g. 0 or 8), the whole `select` statement would be ignored and no part of it executed and the next statement to be executed would be `Print(day)`.

Optionally, a special `case` statement can be added just before the `endselect` keyword. This is the `case default` option which is used to catch all other values which have not been mentioned in previous `case` statements. For example, if we modified our `select` statement above to end with the code

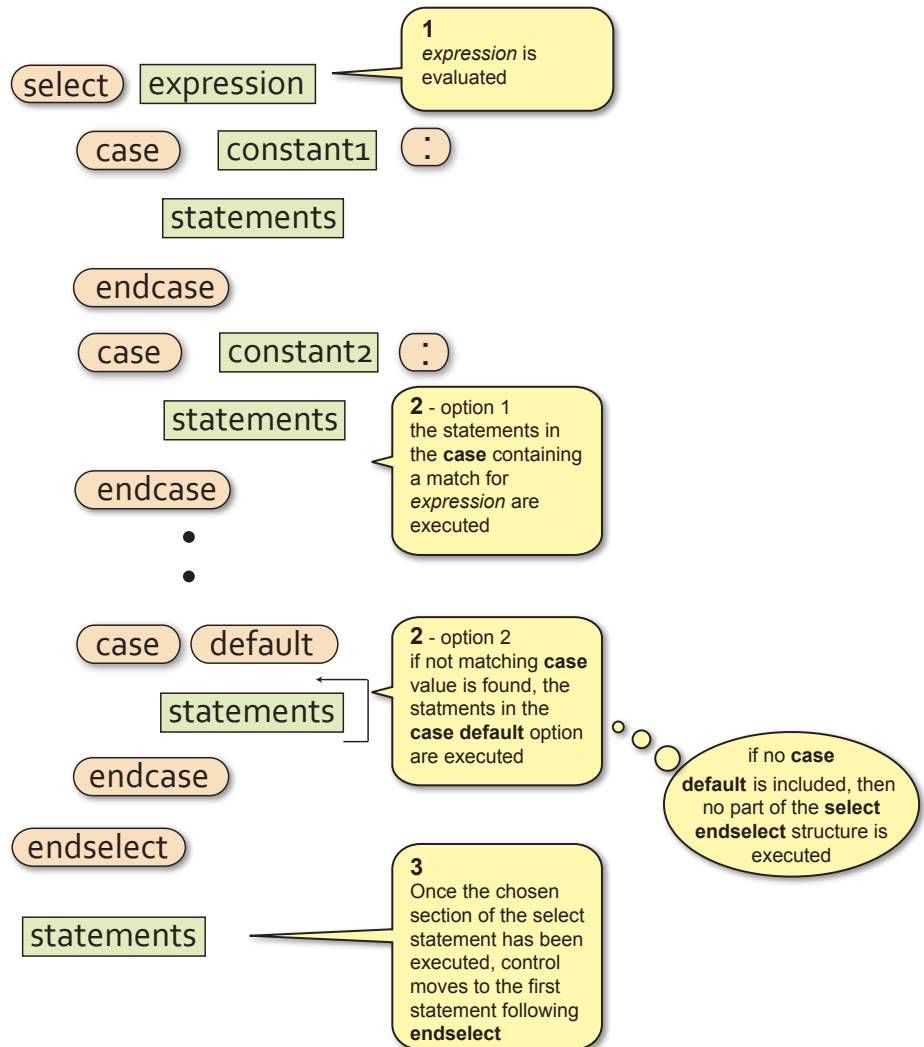
```
case 7:
  Print("Saturday")
endcase
case default
  Print("Invalid day")
endcase
endselect
```

then, if a value outside the range 1 to 7 is generated, the statement in the **case default** option will be executed.

FIG-4.10 shows how the **select** statement is executed.

FIG-4.10

How select Works



Several values can be specified for each **case** option. If the value of the term given in the **select** statement matches any of the values listed in a **case** statement, then the statement(s) in that **case** option will be executed. For example, using the lines

```
num = Random(1,10)
select num
  case 1,3,5,7,9:
    Print("Odd")
  endcase
  case 2,4,6,8,10:
    Print("Even")
  endcase
endselect
print(num)
Sync()
```

the word *Odd* would be displayed if any odd number between 1 and 9 was entered.

The values given beside the **case** keyword may also be a string as in the example below:

```
name$ = GetName()  
select name$  
  case "Jack","Jill" :  
    Print("Hello friend")  
  endcase  
  case default  
    Print("I do not know your name")  
  endcase  
endselect  
Sync()
```

GetName() is assumed to be a user-written function that allows the player to enter their name.

Although the **case** value may also be a real value as in the line

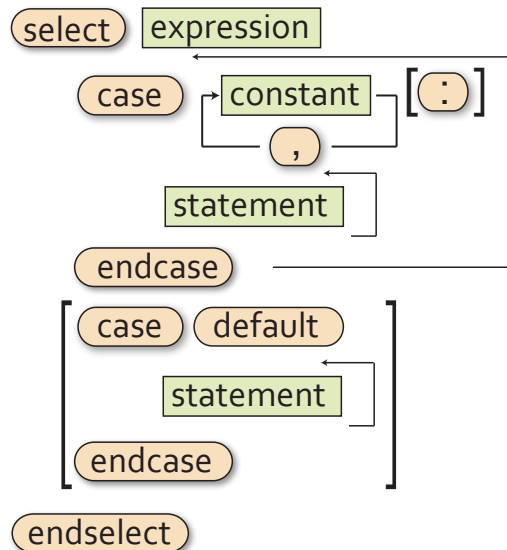
CASE 1.52

it is a bad idea to use these since the machine cannot store real values accurately. If a real variable contained the value 1.52000001 it would not match with the **case** value given above.

The general format of the **select** statement is given in FIG-4.11.

FIG-4.11

select..endselect



where:

- | | |
|-------------------|--|
| expression | is a variable or expression which reduces to a single integer, real or string value. |
| value | is a constant of any type (integer, real or string). |
| statement | is any valid AGK BASIC statement (even another select statement!). |

Activity 4.19

Start a new project, *Days*.

The program should generate a random number in the range 0 to 8 and display the corresponding day of the week if the number is in the range 1 to 7. For any other value, the message *Invalid day* should be displayed.

Test and save your program.

Activity 4.20

Start a new project, *Cards*.

Generate a random number in the range 1 to 13 (the number represents the value of a playing card - 11, 12 and 13 being the Jack, Queen and King).

The program should display the message *Court card* if 11, 12, or 13 is generated and *Spot card* for all other values.

Test and run your program.

Not all multi-way selection situations can be coded using the `select..endselect` statement. For example, let's say a number can be in the range 1 to 1000 and we want to perform specific actions for each of the groupings 1 to 200, 201 to 400, 401 to 600, 601 to 1000 then, since it would be impractical to list all the possible values for each group in a `case` line, we would have to code such a problem using nested `if` statements.

Testing Selective Code

When a program contains one or more `if` structures, our test strategy has to change to cope with this. For every `if` statement within a program we need to create at least two test values: one which results in the condition within the `if` statement being true, the other in the condition being false. Therefore, if a program contained the lines

```
no = GetButtonEntry()
if no mod 2 = 0
    Print("This is an even number")
endif
```

then we need to have a test value for `no` which is even and another which is odd. For example, we could choose the values 10 and 3.

Another important test for conditions involving *less than*, or *greater than* operators is to find out what happens when the variable's value is exactly equal to the value against which it is being tested. For example, if a program contained the lines

```
if result < 0
    Print("Negative")
else
    Print("Positive")
endif
```

then we would want to include zero as one of our test values, giving us three test

This also applies to *less than or equal to* and *greater than or equal to* operators.

values: one less than zero, zero, and one greater than zero. So we could use, say, -7, 0 and 8.

Some of our projects don't allow for user input - instead they use randomly generated values. So we have no control over what values will be used when the program is run!

For test purposes, in a situation like this, we can modify the program's code temporarily so we can control the value used. Hence, in our *Numbers* project, for example, we could change the line

```
no = Random(-12,12)
```

to

```
no = -7
```

Now we can run the program and see if we get the expected result.

In the next two runs of the program we would change the assignment line to 0 and then 8 to get our other two test values. Once we have satisfied ourselves that the expected results have been obtained then we must restore the original code line to the program allowing the value of *no* to be generated randomly once more.

When an **if** statement contains more than one condition linked with **and** or **or** operators, testing needs to check each possible combination of true and false settings. For example, if a program contained the line

```
if dice1 = 6 and dice2 = 6
```

then our tests should include all possible combinations of true and false for the two conditions. A possible set of values is shown in FIG-4.10.

FIG-4.10

Test Data and
Condition Results

dice1	dice2	Result
3	5	false, false
1	6	false, true
6	4	true , false
6	6	true , true

In a complex condition it is sometimes not possible to create every theoretical combination of true and false. For example, if a program contains the line

```
if total = 7 or total = 11 or dice1 = dice2
```

then the combinations of true and false for the three conditions are shown in FIG-4.11.

FIG-4.11

Three Condition
Permutations

total=7	total=11	dice1=dice2
false	false	false
false	false	true
false	true	false
false	true	true
true	false	false
true	false	true
true	true	false
true	true	true

But the last two combinations in the table are impossible to achieve since *total* cannot

contain the values 7 and 11 at the same time (the conditions are mutually exclusive). So our test data will have test values which create only the remaining 6 combinations.

Activity 4.21

Suggest a set of test values for the latest version of the *Dice* project (Activity 4.17).

How would we have to modify the program's code in order to use these test values?

Summary

- The term **nested if statements** refers to the construct where one or more `if` statements are placed within the structure of another `if` statement.
- Multi-way selection can be achieved using nested `if` or by using the `select` statement.
- The `select` statement can be based on integer, real or string values.
- The `case` line can have any number of values, each separated by a comma.
- The `case default` option is executed when the value being searched for matches none of those given in the CASE statements.
- Testing a simple `if` statement should ensure that both true and false results are tested.
- Where a specific value is mentioned in a condition (as in `no < 0`), that value should be part of the test data.
- When a condition contains `and` or `or` operators, every possible combination of results should be tested.
- Nested `if` statements should be tested by ensuring that every possible path through the structure is executed by the combination of test data.
- `select` structures should be tested by using every value specified in the `case` statements.
- `select` should also be tested using a value that does not appear in any of the `case` statements.

Solutions

Activity 4.1

- a) Valid.
- b) Valid.
- c) Valid.
- d) Invalid. \Rightarrow is not a relational operator (should be \geq).
- e) Invalid. Integer variable compared with string.
- f) Invalid. 14 High Street should be in double quotes.

Activity 4.2

- a) False. Only the second string contains a space.
- b) True. "def" is shorter and matches the first three characters of "defg".
- c) True. "A" comes before "B".
- d) False. Only the second string contains a full stop.
- e) False. Only the second string contains a capital D.
- f) True. "*" has a greater ASCII coding than "&".

Activity 4.3

Program code:

```
rem *** include Buttons code ***
#include "Buttons.agc"
rem *** Setup the buttons for input ***
SetUpButtons()
rem *** Get the first value ***
Print("Enter first value :")
Sync()
Sleep(2000)
no1 = GetButtonEntry()
rem *** Get the second value ***
Print("Enter second value : ")
Sync()
Sleep(2000)
no2 = GetButtonEntry()
rem *** if no remainder, display message ***
if no1 mod no2 = 0
    Print("Exactly divisible")
    Sync()
endif
do
loop
```

Activity 4.4

Modified code for *Dice* is:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message if guess is wrong ***
if guess <> dice
    Print("Wrong")
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

Activity 4.5

Modified code for *Dice* is:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message and difference ***
rem *** if guess is wrong ***
if guess <> dice
    PrintC("Wrong. You were out by ")
    difference = dice - guess
    Print(difference)
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

You may get a negative value displayed when the guess is greater than the random number generated.

Activity 4.6

Code for *TwoDice*:

```
rem *** Two dice ***

rem *** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)
rem *** Check for a win ***
total = dice1 + dice2
if total = 7 or total = 11
    Print("You win!")
endif
rem *** Display dice values ***
PrintC("Value of dice 1 : ")
Print(dice1)
PrintC("Value of dice 2 : ")
Print(dice2)
Sync()
do
loop
```

Activity 4.7

Modified code for *TwoDice*:

```
rem *** Two dice ***

rem *** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)
rem *** Check for a win ***
total = dice1 + dice2
if total = 7 or total = 11 or dice1 = dice2
    Print("You win!")
endif
rem *** Display dice values ***
PrintC("Value of dice 1 : ")
Print(dice1)
PrintC("Value of dice 2 : ")
Print(dice2)
Sync()
do
loop
```

Activity 4.8

Code for *ThreeDice*:

```
rem *** Three Dice ***

rem *** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)
dice3 = Random(1,6)
rem *** IF any two dice match THEN ***
if dice1 = dice2 or dice1 = dice3 or dice2 = dice3
    Print("You win!")
endif
rem *** Display values ***
PrintC("dice 1: ")
Print(dice1)
PrintC("dice 2: ")
Print(dice2)
PrintC("dice 2: ")
Print(dice3)
Sync()
do
loop
```

Activity 4.9

- IF player lands on "Go to Jail" OR player picks up a "Go to Jail" card OR player throws three doubles in a row THEN
- IF 10,00 gold pieces collected OR princess freed AND dragon slayed THEN
- IF (holding King of Spades OR holding Queen of Spades) AND Ace of Diamonds played THEN

Activity 4.10

```
dice1 <> dice2 and dice1 <> dice3 and dice2 <> dice3
```

Activity 4.11

Modified code for *Dice* is:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message ***
if guess = dice
    Print("Correct")
else
    Print("Wrong")
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

Activity 4.12

Code for *TwoNumbers*

```
rem *** Smaller odd/even ***

rem *** include Buttons functions ***
#include "Buttons.agc"
```

```
rem *** Display buttons ***
SetUpButtons()
rem *** Get numbers ***
Print("Enter first number ")
Sync()
Sleep(2000)
no1 = GetButtonEntry()
Print("Enter second number ")
Sync()
Sleep(2000)
no2 = GetButtonEntry()
rem *** Determine smaller value ***
if no1 < no2
    answer = no1
else
    answer = no2
endif
rem *** Display smaller ***
PrintC("Smaller value is ")
Print(answer)
rem *** Determine if answer is odd or even ***
if answer mod 2 = 0
    Print("This is an even number")
else
    Print("This is an odd number")
endif
Sync()
do
loop
```

Activity 4.13

- A Boolean expression is an expression whose result is either true or false.
- Six. <, <=, >, >=, =, <>
- not is performed first, and next and or last. This order will change if parentheses are used.

Activity 4.14

Modified code for *Dice* is:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message ***
if guess = dice
    Print("Correct")
else
    if guess > dice
        Print("Your guess is too high")
    else
        Print("Your guess is too low")
    endif
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

Activity 4.15

Code for *Number*:

```
rem *** Random number between -12 and 12 ***

rem *** Generate number ***
no = Random(-12,12)
rem *** Display number's sign ***
```

```

if no < 0
    Print("Negative")
else
    if no = 0
        Print("Zero")
    else
        Print("Positive")
    endif
endif
rem *** Display number ***
Print(no)
Sync()
do
loop

```

Activity 4.16

Modified code for *Dice*:

```

rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
rem *** Display message ***
diff = dice - guess
if diff > 2
    Print("Your guess is too low")
else
    if diff > 0
        Print("Your guess is slightly too low ")
    else
        if diff = 0
            Print("Correct")
        else
            if diff >= -2
                Print("Your guess is slightly too
                    high")
            else
                Print("Your guess is too high")
            endif
        endif
    endif
endif
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop

```

Activity 4.17

The multi-way selection section of *Dice*'s code should now be have the following layout:

```

if diff > 2
    Print("You guess is too low")
else if diff > 0
    Print("Your guess is slightly too low ")
else if diff = 0
    Print("Correct")
else if diff >= -2
    Print("Your guess is slightly too high")
else
    Print("Your guess is too high")
endif endif endif endif

```

Activity 4.18

New new multi-way selection coding in *Dice* should now be:

```

if diff > 2
    Print("You guess is too low")

```

```

elseif diff > 0
    Print("Your guess is slightly too low ")
elseif diff = 0
    Print("Correct")
elseif diff >= -2
    Print("Your guess is slightly too high")
else
    Print("Your guess is too high")
endif

```

Activity 4.19

Code for *Days*:

```

rem *** Display day of the week ***

rem *** Generate value ***
day = Random(0,8)

rem *** Display day of week ***
select day
case 1:
    Print("Sunday")
endcase
case 2:
    Print("Monday")
endcase
case 3:
    Print("Tuesday")
endcase
case 4:
    Print("Wednesday")
endcase
case 5:
    Print("Thursday")
endcase
case 6:
    Print("Friday")
endcase
case 7:
    Print("Saturday")
endcase
case default
    Print("Invalid day")
endcase
endselect
rem *** Display number generated ***
Print(day)
Sync()
do
loop

```

Activity 4.20

Code for *Cards*:

```

rem *** Cards ***

rem *** Generate card value ***
card = Random(1,13)

rem *** Display card type ***
select card
case 11,12,13:
    Print("Court card")
endcase
case default
    Print("Spot card")
endcase
endselect
Print(card)
Sync()
do
loop

```

Note that all of the spot cards can be handled in the `case default` option because there is no chance of an invalid value being used.

Activity 4.21

The test data needs to cover all the possible paths through the nested if statements. In doing this we will have tested each condition for both true and false options.

So possible values are

dice	guess	Expected results
8	2	Your guess is too low
5	4	Your guess is slightly too low
7	7	Correct
2	4	Your guess is slightly too high
3	8	Your guess is too high

In addition, we would expect the values of dice and guess to be displayed.

Since the dice values are randomly generated it would be impractical to use our test data. We can overcome this problem by setting the variable dice to a specific value rather than determining its value using `Random()`. Once testing is complete, the random assignment can be restored.

5

Iteration

In this Chapter:

- ☐ **while..endwhile** Structure
- ☐ **repeat..until** Structure
- ☐ **for..next** Structure
- ☐ **do..loop** Structure
- ☐ Validating Input
- ☐ The **exit** Statement
- ☐ Testing Loop Structures

Iteration

Introduction

Iteration is the term used when one or more statements are carried out repeatedly. As we saw in Chapter 1, structured English has three distinct iterative structures: FOR .. ENDFOR, REPEAT .. UNTIL and WHILE .. ENDWHILE.

AGK BASIC, on the other hand, has four iterative structures. One of these takes the same form as their structured English equivalent, but others differ slightly and therefore care should be taken when translating structured English statements to AGK BASIC.

The `while..endwhile` Construct

The `while` statement is probably the easiest of AGK BASIC's loop structures to understand, since it is identical in operation and syntax to the WHILE loop in structured English.

This structure allows us to continually execute a section of code as long as a specified condition is being met. For example, if, in a game, a player's character sustains damage of 10 points while he stands on a "bad health" area, this can be described in structured English as

```
WHILE player on "bad health" area DO
  Reduce player's health by 10
ENDWHILE
```

which can be coded in AGK BASIC as:

The code assumes a variable called `floor_area` records the position of the character and that the "bad health" area is at position 25.

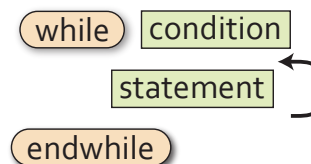
```
while floor_area = 25
  health = health - 10
endwhile
```

The syntax of AGK BASIC's `while .. endwhile` construct is shown in FIG-5.1.

FIG-5.1

`while..endwhile`

AGK BASIC's `while` statement does not use the term `do`.



where:

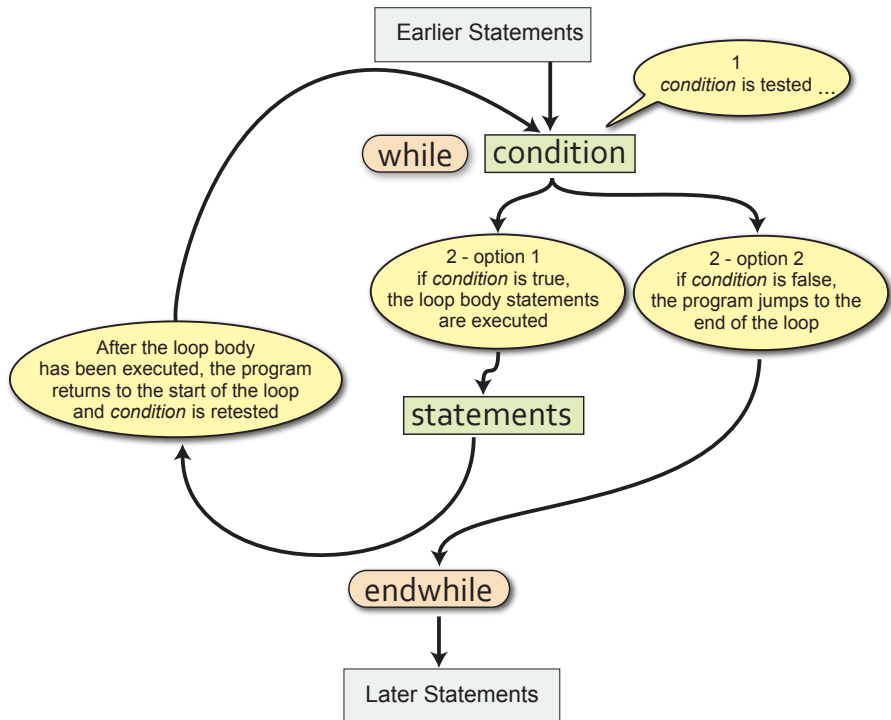
condition	is a Boolean expression and may include <code>and</code> , <code>or</code> , <code>not</code> and parentheses as required.
statement	is any valid AGK BASIC statement.

`while..endwhile` is an **entry-controlled** loop. That is, the condition at the start of the loop is tested and only if that condition is true, are the statements within the loop executed. When the `endwhile` term is reached, control returns to the `while` line and the condition is retested. If the condition is found to be false, then looping stops with an immediate jump from the `while` line to the `endwhile` line, skipping the statements in between.

A visual representation of how this loop operates is shown in FIG-5.2.

FIG-5.2

How **while..
endwhile** Operates



Note that the loop body may never be executed if *condition* is false when first tested.

A common use for this loop statement is validation of input. So, for example, in our number guessing game, we might ensure that the user types in a value between 0 and 9 when entering their guess by using the logic

```
Get guess
WHILE guess outside the range 0 to 9 DO
    Display error message
    Get guess
ENDWHILE
```

which can be coded in AGK BASIC using our *GetButtonEntry()* function as:

```
Print("Enter your guess (0 - 9) : ")
Sync()
Sleep(2000)
guess = GetButtonEntry()
while guess < 0 or guess > 9
    Print("Your guess must be between 0 and 9")
    Print("Enter your guess again(0 - 9) : ")
    Sync()
    Sleep(2000)
    guess = GetButtonEntry()
endwhile
```

The test `guess < 0` is not required since the function `GetButtonEntry()` does not allow negative values to be entered. However, the condition has been included so that, should `GetButtonEntry()` ever be modified to allow entry of negative values, the `while` loop will catch any values less than zero.

Activity 5.1

Modify your *Dice* project to incorporate the code given above. Check that the program works correctly by attempting to make guesses which are outside the range 0 to 9. Resave your project.

Activity 5.2

A simple dice game involves counting how many times in a row a pair of dice can be thrown to produce a value of 8 or less. The game stops as soon as a value greater than 8 is thrown.

Create a new project, *DiceCount*, which implements the following logic:

```
Set count to zero
Throw the two dice
Display dice values
WHILE the sum of the two dice <= 8 DO
    Add 1 to count
    Throw the two dice
    Display dice values
ENDWHILE
Display "You had a run of " , count, "throws"
```

Test and save your program.

The repeat...until Construct

Like structured English, AGK BASIC has a `repeat...until` statement. The two structures are identical. Hence, if in structured English we write

```
Set total to zero
REPEAT
    Get a number
    Add number to total
UNTIL number is zero
```

then the same logic would be coded in AGK BASIC as

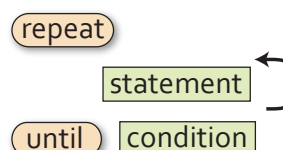
```
total = 0
repeat
    number = GetButtonEntry()
    total = total + number
until number = 0
```

The code assumes we are using the Button routines introduced in the previous chapter to accept input.

The `repeat...until` statement is an **exit-controlled** loop structure. That is, the action within the loop is executed and then an exit condition is tested. If that condition is found to be true, then looping stops, otherwise the statements specified within the loop are executed again. Iteration continues until the exit condition is true. The syntax of the REPEAT statement is shown in FIG-5.3.

FIG-5.3

repeat...until



where:

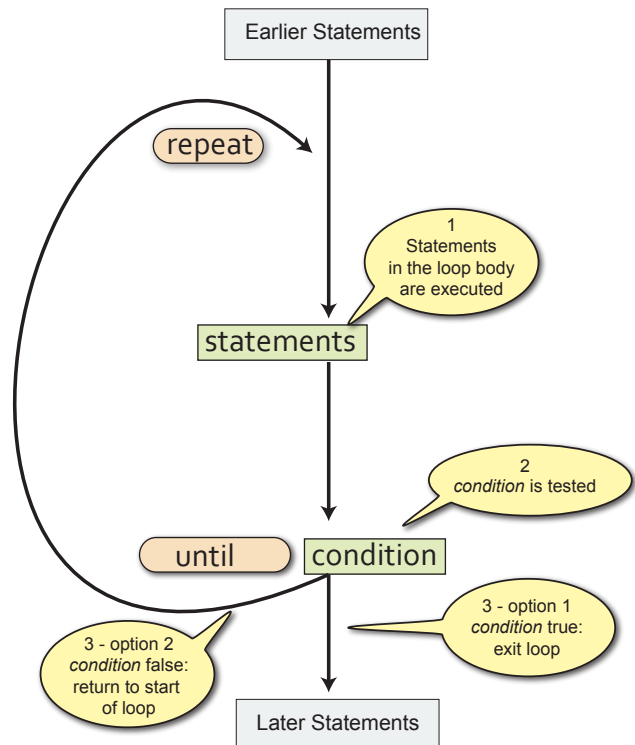
condition is a Boolean expression and may include **and**, **or**, **not** and parentheses as required.

statement is any valid AGK BASIC statement.

The operation of the **repeat .. until** construct is shown graphically in FIG-5.4.

FIG-5.4

How **repeat..until**
Operates



Activity 5.3

Create a new project, *Total*, to read in a series of integer values, stopping only when a zero is entered. The values entered should be totalled and that total displayed at the end of the program. Use the Buttons routines to accept input.

Use the following logic:

```
Set total to zero
REPEAT
    Get a number
    Add number to total
UNTIL number is zero
Display total
```

Test and save your project.

Activity 5.4

Modify *Dice* to allow the player to keep guessing until the correct number is arrived at.

Test and save your project.

The `for..next` Construct

In structured English, the FOR loop is used to perform an action a specific number of times. For example, we might describe dealing seven cards to a player using the logic:

```
FOR 7 times DO
  Deal card
ENDFOR
```

Sometimes the number of times the action is to be carried out is less explicit. For example, if each player in a game is to pay a £10 fine, we could write:

```
FOR each player DO
  Pay £10 fine
ENDFOR
```

However, in both examples, the action specified between the FOR and ENDFOR terms will be executed a known number of times.

In AGK BASIC the `for` construct makes use of a variable to keep a count of how often the loop is executed and the first line of the structure takes the form:

```
for variable = start_value to finish_value
```

Hence, if we want a `for` loop to iterate 7 times we would write

```
for c = 1 to 7
```

In this case *c* would be assigned the value 1 when the `for` loop is about to start. Each time the statements within the loop are completed, *c* will be incremented, and eventually, when *c* is equal to 7 and the loop body has been executed, iteration stops.

The variable used in a `for` loop is known as the **loop counter**.

Activity 5.5

Write the first line of a `for` loop that is to be executed 10 times, using a variable *j* as the loop counter. The starting value of *j* should be 1.

While structured English marks the end of a FOR loop using the term ENDFOR, in AGK BASIC the end of the loop is indicated by the term `next` followed by the name of the loop counter variable used in the `for` statement. For example, the code

```
for k = 1 to 10
  Print("*")
next k
Sync()
```

contains a single statement within the loop body and will display a column of 10 asterisks.

Activity 5.6

What would be displayed by the code

```
for p = 1 to 10
    Print(p)
next p
Sync()
```

The loop counter in a `for` loop can be made to start and finish at any value, so it is quite valid to start a loop with the line:

```
for m = 3 to 12
```

The loop counter *m* will contain the value 3 when the loop is first executed and 12 when the final execution is complete. The loop will be executed exactly 10 times.

If the start and finish values are identical, as in the line

```
for r = 10 to 10
```

then the loop is executed once only.

Where the start value is greater than the finish value, the loop will not be executed at all so the code within the loop body will be ignored. Such a result would be produced from the line

```
for k = 10 to 9
```

Normally, 1 is added to the loop counter each time the loop body is performed. However, we can change this by adding a `step` value to the `for` loop as in the example shown below:

```
for c = 2 to 10 step 2
```

In this last example the loop counter, *c*, will start at 2 and then increment to 4 on the next iteration. The program in FIG-5.5 uses the `step` option to display the 7 times table from 1 x 7 to 12 x 7.

FIG-5.5

7 Times Table

```
rem *** 7 Times Table ***

rem *** Display title ***
Print("7 Times Table")
Print("")
rem *** Display the table values ***
for c = 7 to 84 step 7
    Print(c)
next c
Sync()
do
loop
```

Activity 5.7

Start a new project, *Tables*, that implements the code shown in FIG-5.5.

Test the program.

Modify the program so that it displays the 12 times table from 1 x 12 to 12 x 12.

By using the **step** keyword with a negative value, it is even possible to create a **for** loop that reduces the loop counter on each iteration as in the line:

```
for d = 10 to 0 step -1
```

This last example causes the loop counter to start at 10 and finish at 0.

Activity 5.8

Modify *Tables* so that the 12 times table is displayed with the highest value first. That is, starting with 144 and finishing with 12.

It is possible that the **step** value given may cause the loop counter never to match the finish value. For example, in the line

```
for c = 1 to 12 step 5
```

the variable *c* will take on the values 1, 6, and 11. The loop body will not be executed when the loop counter passes the finishing value (12, in this case) and the looping will stop.

The start, finish and even step values of a **for** loop can be defined using a variable or arithmetic expression, as well as a constant. For example, in FIG-5.6 below the user is allowed to enter the upper limit of the **for** loop.

FIG-5.6

Using a Variable in a
for..next Statement

```
#include "Buttons.agc"

SetUpButtons()
rem *** Get a number ***
Print("Enter upper limit")
Sync()
Sleep(2000)
num = GetButtonEntry()
rem *** Display values between 1 and num ***
for c = 1 to num
    Print(c)
    Sync()
    Sleep(200)
next c
do
loop
```

The program will display every integer value between 1 and the number entered by the user. If this involves many numbers being displayed, there will not be space within the app window to show them all at the same time. Therefore, the program displays one number at a time with 0.2 secs delay between each value.

Activity 5.9

Start a new project, *OneTo*, containing the code given in FIG-5.6. (Remember you have to include the three *Buttons* files in your project folder).

Modify the program so that the user may also specify the starting value of the **for** loop.

Change the program a second time so that the user can specify a step size for the **for** loop.

Test each version of the program.

The **for** loop counter can also be specified as a real value with a **step** value which is not a whole number. For example:

```
for ch# = 1.0 to 2.0 step 0.1
  Print(ch#)
next ch#
Sync()
```

Activity 5.10

Create a project, *ForReal*, which includes the code given above and check out the result.

►► The latest version of AGK no longer displays values to 11 decimal places; only 6, so the rounding errors are no longer visible but still occur internally.

Notice that most of the values displayed by the last Activity are slightly out. For example, instead of the second value displayed being 1.1, it displays as 1.10000002384.

This difference is caused by rounding errors when converting from the decimal values that we use to the binary values favoured by the computer.

Although we might have expected the **for** loop to perform 11 times (1.0, 1.1, 1.2, etc. to 2.0), in fact, it only performs 10 times up to 1.90000021458. Again, this discrepancy is caused by the rounding error problem.

Activity 5.11

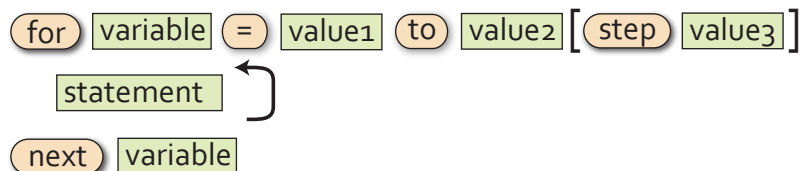
Modify *ForReal* so that the upper limit of the loop is 2.01.

How many times is the iteration performed now?

The format of the **for...next** construct is shown in FIG-5.7.

FIG-5.7

for...next



where:

variable

is either an integer or real variable. Both *variable* tiles in the diagram refer to the same variable. Hence, the name used after

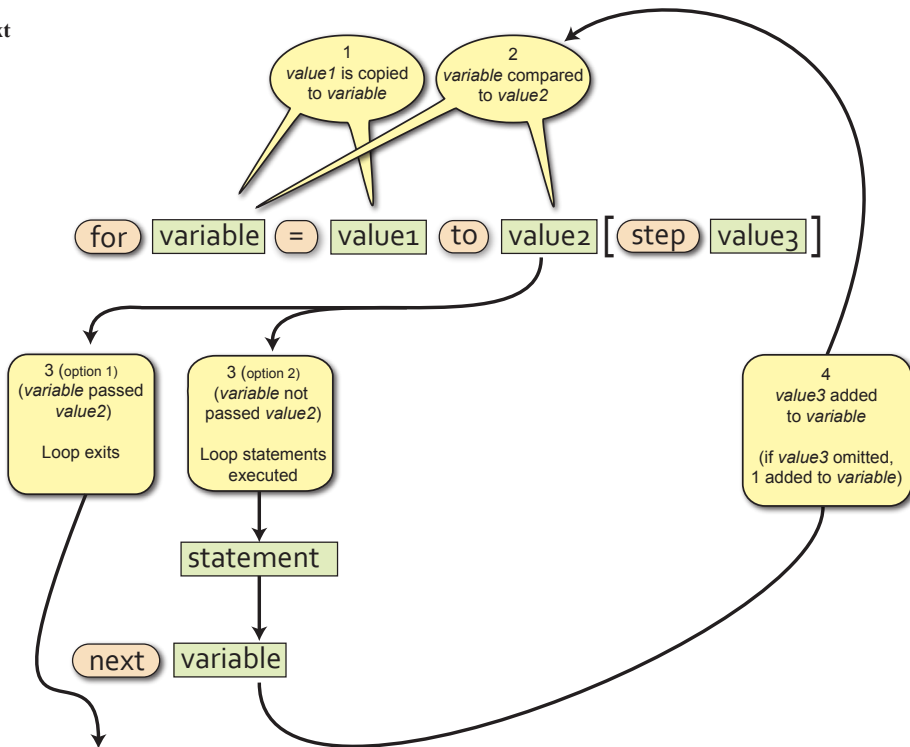
the keywords **for** and **next** must be the same. This variable is known as the **loop counter**.

- value1** is the initial value of the loop counter. The loop counter will contain this value the first time the statements within the loop are executed.
- value2** is the final value of the loop variable. The loop variable will usually contain this value the last time the loop body is executed.
- value3** is the value to be added to the loop counter after each iteration. If this is omitted then a value of 1 is added to the loop counter.
- statement** is any valid AKG BASIC statement.

The operation of the **for..next** statement is shown graphically in FIG-5.8.

FIG-5.8

How **for..next**
Operates



Activity 5.12

Create a new project, *InTotal*, which reads in and displays the total of 6 numbers. Make use of the *Buttons* files for input.

Test and save your project.

Activity 5.13

Start a new project called *Shades*.

Code a program which uses a `for` loop with a start value of 0 and finish of 255.

Inside the loop, execute a `SetColor()` statement and use the value of the loop counter as the red parameter to the statement. The green and blue parameter values for the `SetColor()` statement should both be zero.

Add a delay (using `sleep()`) of 20 milliseconds between each iteration of the loop.

Test and save your project.

Finding the Smallest Value in a List of Values

There are several tasks that will crop up over and over again in your programs. One of these is finding the smallest value in a list of numbers. This is a trivial enough task for our own brains as long as the list is short enough to be taken in at a glance, but if asked how you managed to come up with the correct answer, you might struggle to give a verbal description of the strategy you used.

Now, let's imagine you wanted to record the coldest temperature achieved in your area during the current year. Since this involves a longer list of data which also takes a full year to access, you would have to come up with an organised way of getting the information you want. Perhaps you would write down the lowest temperature on January 1st and then check each day to see if a lower temperature has been achieved. When a lower temperature does occur, you can erase the previous record and write down this new temperature. By the end of the year your record would show the lowest temperature achieved during the year.

This is exactly how we tackle the same type of problem in a computer program. We set up one variable to hold the smallest value we've come across so far and if a later value is smaller, it is copied into this variable. The algorithm used is given below and assumes 7 numbers will be entered in total:

```
Get first number
Set smallest to first number
FOR 6 times DO
    Get next number
    IF number < smallest THEN
        Set smallest to number
    ENDIF
ENDFOR
Display smallest
```

Activity 5.14

Create a new project called *Smallest*.

In this program implement the logic shown above to display the smallest of 5 integer values entered.

Modify the program to find the largest, rather than the smallest, of the numbers entered. Save your project.

The exit Statement

The `exit` statement is used to terminate the loop currently being executed. The next statement to be executed after an `exit` command is the statement immediately after the end of the loop. The `exit` statement takes the form shown in FIG-5.9.

FIG-5.9

The `exit` Statement

`exit`

Normally, the `exit` statement will appear within an `if` statement.

Let's look at an example where the `exit` statement might come in useful. In a dice game we are allowed to throw a pair of dice 5 times and our score is the total of the five throws. However, if during our throws we throw a 1, then, according to the rules of the game, our turn ends and our final score becomes the total achieved up to that point (excluding the throw containing a 1). We could code this game as shown in FIG-5.10.

FIG-5.10

Using `exit`

```
rem *** set total to zero ***
total = 0
rem *** for 5 times do ***
for c = 1 to 5
    rem *** Display roll number ***
    PrintC("Roll number ")
    Print(c)
    Sync()
    Sleep(1000)
    rem *** throw both dice ***
    dice1 = Random(1,6)
    dice2 = Random(1,6)
    rem *** display throw number and dice values ***
    PrintC("dice 1 : ")
    PrintC(dice1)
    PrintC("           dice 2 : ")
    Print(dice2)
    Sync()
    Sleep(4000)
    rem *** if either dice is a 1 then quit loop ***
    if dice1 = 1 or dice2 = 1
        exit
    endif
    rem *** add dice throws to total ***
    total = total + dice1 + dice2
next c
rem *** display final score ***
PrintC("your final score was : ")
Print(total)
Sync()
do
loop
```

Activity 5.15

Create a new project call *SumDice*. Delete the existing code in *main.agc* and enter the program given in FIG-5.10.

Run the program and check that the loop exits if a 1 is thrown.

Modify the program to exit only if both dice show a 1.

The do .. loop Construct

The `do .. loop` construct is a rather strange loop structure, since, while other loops are designed to terminate eventually, the `do .. loop` structure will continue to repeat the code within its loop body indefinitely.

The default code that exists when we begin a new project makes use of this loop structure to continually display the words *Hello world* - the traditional text for a first program.

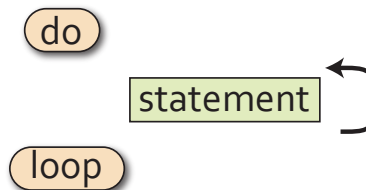
When a `do` loop is executing, then, under normal circumstances, the program will only terminate when forced to do so by an external event. In all our projects so far, the external event has been the operating system closing down our program in response to our clicking on the X button at the top-right of the app window. Alternatively, an `exit` statement can be included within the loop to allow the loop to be exited when a given condition occurs.

As we write more complex programs you will begin to understand why a `do` loop is so often needed to get the game to run smoothly.

The `do .. loop` structure takes the format shown in FIG-5.11.

FIG-5.11

do..loop



Nested Loops

A common requirement within a program is to place one loop control structure within another. This is known as **nested loops**. For example, to input six game scores (each between 0 and 100) and then calculate their average, the logic required is:

1. Set total to zero
2. FOR 6 times DO
3. Get valid score
4. Add score to total
5. ENDFOR
6. Calculate average as total / 6
7. Display average

This appears to have only a single loop structure beginning at statement 2 and ending at statement 5. However, if we add detail to statement 3, this gives us

3. Get valid score
 - 3.1 Read score
 - 3.2 WHILE score is invalid DO
 - 3.3 Display "Score must be between 0 to 100"
 - 3.4 Read score
 - 3.5 ENDWHILE

which, if placed in the original solution, results in a nested loop structure, where a `while` loop appears inside a `for` loop:

1. Set total to zero
2. FOR 6 times DO
- 3.1 Read score

```

3.2   WHILE score is invalid DO
3.3       Display "Score must be between 0 to 100"
3.4       Read score
3.5   ENDWHILE
4.     Add score to total
5.   ENDFOR
6.   Calculate average as total / 6
7.   Display average

```

Activity 5.16

Turn the above algorithm into an AGK BASIC project, *AverageScore*, using the *Buttons* files to allow input.

Run and test the program, making sure it operates as expected.

Nested for Loops

A very common example of nested loops are nested `for` loops. And, although someone new to programming can sometimes have difficulties with the concept, it's actually easy enough to see real world examples of how nested `for` loops work.

Next time you are out in the car, have a look at the odometer (that's the one that tells you how many miles/kilometres the car has done). Now, look at the right two digits of the odometer. As you travel along you'll see the far right hand digit move slowly until it reaches 9; at that point it returns to zero and the digit to its left increments before the whole process repeats itself. You'll see the same sort of thing on a digital clock.

The code in FIG-5.12 emulates those last two digits on the odometer. Initially, they are set to 00 and then move onto 01, 02 ... 09, 10, 11, etc

FIG-5.12

Nested `for` loops

```

Rem *** Nested for loop ***
for tens = 0 to 9
    for units = 0 to 9
        PrintC(tens)
        PrintC(" ")
        Print(units)
        Sync()
        Sleep(200)
    next units
next tens
do
loop

```

The *tens* loop is known as the **outer loop**, while the *units* loop is known as the **inner loop**.

A few points to note about nested `for` loops:

- The inner loop increments fastest.
- Only when the inner loop is complete does the outer loop variable increment.
- The inner loop counter is reset to its starting value each time the outer loop counter is incremented.

Activity 5.17

Start a new project, *NestedFor*, and code the program to match FIG-5.12. Test and save your project.

Activity 5.18

What would be output by the following code?

```
for no1 = -2 to 1
  for no2 = 0 to 3
    PrintC(no1)
    PrintC(" ")
    Print(no2)
    Sync()
    Sleep(200)
  next no2
next no1
```

Testing Iterative Code

We need a test strategy when looking for errors in iterative code. Where possible, it is best to create at least three sets of values:

- Test data that causes the loop to execute zero times.
- Test data that causes the loop to execute once.
- Test data that causes the loop to execute multiple times.

For example, in *Dice* we added statements to ensure that the guess entered was in the range 0 to 9 using the following code:

```
guess = GetButtonEntry()
while guess < 0 or guess > 9
  Print("Your guess must be between 0 and 9")
  Print("Enter your guess again(0 - 9) : ")
  Sync()
  Sleep(2000)
  guess = GetButtonEntry()
endwhile
```

To test the `while` loop in this code we could use the test data shown in FIG-5.13.

FIG-5.13

Test Data

Test No.	guess
1	7
2	10, 5
3	18, 12, 3

The `while` loop is only executed if *guess* is outside the range 0 to 9, so Test 1, which uses a value inside that range, will skip the `while` loop body giving zero iterations.

Test 2 starts with an invalid value (10) for *guess*, causing the `while` loop body to be executed, and then uses a valid value (5). This loop is therefore exited after only one iteration.

Test 3 uses two invalid values (18 and 12) before entering a valid value (3), causing the `while` loop body to execute twice.

Activity 5.19

The following code is meant to calculate the average of a sequence of numbers. The sequence ends when the value zero is entered. This terminating zero is not considered to be one of the numbers in the sequence.

```
total = 0
count = 0
Print("Enter number (0 to stop)")
Sync()
Sleep(2000)
num = GetButtonEntry()
while num <> 0
    total = total + num
    count = count + 1
    Print("Enter number (0 to stop)")
    Sync()
    Sleep(2000)
    num = GetButtonEntry()
endwhile
average = total / count
PrintC("Average is ")
Print(average)
Sync()
do
loop
```

Make up a set of test values (similar in construct to FIG-5.13) for the `while` loop in the code.

Create a new project, *Average*, containing the code given above and use the test data to find out if the code functions correctly.

There will be cases where using all three tests strategies are not possible. For example, a `repeat` loop cannot execute zero times and, in this case, we have to satisfy ourselves with single and multiple iteration tests.

A `for` loop, when written for a fixed number of iterations can only be tested for that number of iterations. So a loop beginning with the line

```
for c = 1 to 10
```

can only be tested for multiple iterations (10 iterations, in this case), the exception being if the loop body contains an `exit` statement, in which case zero and one iteration tests may also be possible by supplying values which cause the `exit` statement to be terminated during the required iteration.

A `for` loop which is coded with a variable upper limit as in

```
for c = 1 to max
```

may be fully tested by making sure *max* has the values 0, 1, and more than 1 during testing.

Summary

- AGK BASIC contains four iteration constructs:

```
while .. endwhile
repeat .. until
for .. next
do .. loop
```

- The `while..endwhile` construct executes a minimum of zero times and exits when the specified condition is false.
- The `repeat..until` construct executes at least once and exits when the specified condition is true.
- The `for..next` construct is used when iteration has to be done a specific number of times.
- A step size may be included in the `for` statement. The value specified by the step term is added to the loop counter on each iteration.
- If no step size is given in the `for` statement, a value of 1 is used.
- `for` loops counters can be integer or real.
- The start, finish and step values in a `for` loop can be defined using variables or arithmetic expressions.
- If the start value is equal to the finish value, a `for` loop will execute only once.
- If the start value is greater than the finish value and the `step` size is a positive value, a `for` loop will execute zero times.
- Using the `do..loop` structure creates an infinite loop.
- The `exit` statement can be used to exit from any loop.
- One loop structure can be placed within another loop structure. Such a structure is known as a nested loop.
- Loops should be tested by creating test data for zero, one and multiple iterations during execution whenever possible.

Solutions

Activity 5.1

Modified code for *Dice*:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
while guess < 0 or guess > 9
    Print("your guess must be between 0 and 9")
    Print("Enter your guess again(0 - 9) : ")
    Sync()
    Sleep(2000)
    guess = GetButtonEntry()
endwhile
rem *** Display message ***
diff = dice - guess
if diff > 2
    Print("You guess is too low")
else
    if diff > 0
        Print("Your guess is slightly too low ")
    else
        if diff = 0
            Print("Correct")
        else
            if guess > -2
                Print("Your guess is slightly too
                high")
            else
                Print("Your guess is too high")
            endif
        endif
    endif
endif
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

Activity 5.2

Code for *DiceCount*:

```
rem *** Count dice run ***

rem *** Set count to zero ***
count = 0
rem *** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)
rem *** display dice values ***
PrintC(dice1)
PrintC(" ")
PrintC(dice2)
Sync()
Sleep(500)
rem *** Keep going while total is less than 9 ***
while dice1 + dice2 <= 8
    rem *** add 1 to count ***
    count = count + 1
    rem *** Throw dice ***
    dice1 = Random(1,6)
    dice2 = Random(1,6)
    rem *** display dice values ***
    PrintC(dice1)
    PrintC(" ")
    PrintC(dice2)
```

```
Sync()
Sleep(500)
endwhile
PrintC("You had a run of ")
PrintC(count)
Print(" throws")
Sync()
do
loop
```

Activity 5.3

Set the app window dimensions to 768 wide by 1024 high.

Code for *Total*:

```
rem *** Total a sequence of numbers ***

rem *** include Buttons routines ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()
rem *** Set total to zero ***
total = 0
rem *** Keep going until zero entered ***
repeat
    rem *** Get value ***
    no = GetButtonEntry()
    rem *** Add value to total ***
    total = total + no
until no = 0
rem *** Display total ***
PrintC("Total = ")
Print(total)
Sync()
do
loop
```

Activity 5.4

Modified code for *Dice* (remember to indent all the code between the **repeat** and **until** terms):

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
repeat
    rem *** Display prompt ***
    Print("Guess what my number is ")
    Sync()
    Sleep(2000)
    rem *** Get a value ***
    guess = GetButtonEntry()
    while guess < 0 or guess > 9
        Print("your guess must be between 0 and 9")
        Print("Enter your guess again(0 - 9) : ")
        Sync()
        Sleep(2000)
        guess = GetButtonEntry()
    endwhile
    rem *** Display message ***
    diff = dice - guess
    if diff > 2
        Print("You guess is too low")
    else if diff > 0
        Print("Your guess is slightly too low ")
    else if diff = 0
        Print("Correct")
    else if diff >= -2
        Print("Your guess is slightly too high")
    else
        Print("Your guess is too high")
    endif endif endif
until guess = dice

rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
```

```
Print(guess)
Sync()
do
loop
```

Activity 5.5

```
for j = 1 to 10
```

Activity 5.6

This code would display the values 1 to 10.

Activity 5.7

Modified code for *Tables* (12 times table):

```
rem *** 12 Times Table ***

rem *** Display title ***
Print("12 Times Table ")
Print("")
rem *** Display the table values ***
for c = 12 to 144 step 12
    Print(c)
next c
Sync()
do
loop
```

Activity 5.8

Modified version of *Tables*:

```
rem *** 12 Times Table ***

rem *** Display title ***
Print("12 Times Table ")
Print("")
for c = 144 to 12 step -12
    Print(c)
next c
Sync()
do
loop
```

Activity 5.9

Code for *OneTo*:

```
rem *** Display all values in a range ***

rem *** include Buttons functions ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()
rem *** Get limit ***
Print("Enter the upper limit")
Sync()
Sleep(2000)
num = GetButtonEntry()
rem *** Display numbers 1 to num ***
for c = 1 to num
    Print(c)
    Sync()
    Sleep(200)
next c
do
loop
```

Start value version of *OneTo*:

```
rem *** Display all values in a range ***

rem *** include Buttons functions ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()
rem *** Get lower limit ***
Print("Enter the lower limit")
Sync()
Sleep(2000)
start = GetButtonEntry()
```

```
rem *** Get upper limit ***
Print("Enter the upper limit")
Sync()
Sleep(2000)
num = GetButtonEntry()
rem *** Display numbers start to num ***
for c = start to num
    Print(c)
    Sync()
    Sleep(200)
next c
do
loop
```

Step size version of *OneTo*:

```
rem *** Display values in a range ***

rem *** include Buttons functions ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()
rem *** Get lower limit ***
Print("Enter the lower limit")
Sync()
Sleep(2000)
start = GetButtonEntry()

rem *** Get upper limit ***
Print("Enter the upper limit")
Sync()
Sleep(2000)
num = GetButtonEntry()
rem *** Get step size ***
Print("Enter the step size")
Sync()
Sleep(2000)
increment = GetButtonEntry()
rem *** Display numbers start to num ***
for c = start to num step increment
    Print(c)
    Sync()
    Sleep(200)
next c
do
loop
```

Activity 5.10

Code for *ForReal*:

```
rem *** Display values from 1 to 2 ***
for ch# = 1.0 to 2.0 step 0.1
    Print(ch#)
    Sync()
    Sleep(200)
next ch#
do
loop
```

Notice that the values displayed are 1.0 to 1.9.

Activity 5.11

Modified version of *ForReal*:

```
rem *** Display values from 1 to 2 ***
for ch# = 1.0 to 2.1 step 0.1
    Print(ch#)
    Sync()
    Sleep(200)
next ch#
do
loop
```

The display now runs from 1.0 to 2.0.

Activity 5.12

Code for *InTotal*:

```
rem *** Total input values ***

rem *** Include button functions ***
#include "Buttons.agc"
```

```

rem *** Set up buttons ***
SetUpButtons()
rem *** Set total to zero ***
total = 0
rem *** Read and sum 6 numbers ***
for c = 1 to 6
    Print("Enter number")
    Sync()
    Sleep(1000)
    no = GetButtonEntry()
    total = total + no
next c
PrintC("Total = ")
Print(total)
Sync()
do
loop

```

```

rem *** Get next number ***
Print("Enter number ")
Sync()
Sleep(1000)
no = GetButtonEntry()
rem *** If number larger, record it ***
if no > largest
    largest = no
endif
next c
rem *** Display largest value ***
PrintC("Largest value entered was ")
Print(largest)
Sync()
do
loop

```

Activity 5.13

Code for *Shades*:

```

rem *** Display all shades of red ***
rem *** Set red intensity to ***
rem *** range from 0 to 255
for red = 0 to 255
    SetClearColor(red,0,0)
    Sync()
    Sleep(20)
next red
do
loop

```

Activity 5.14

Code for *Smallest*:

```

rem *** Find Smallest Number Entered ***

rem *** Include Button functions ***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Get first number ***
Print("Enter number ")
Sync()
Sleep(2000)
no = GetButtonEntry()
rem *** Set smallest to first number ***
smallest = no
rem *** FOR 4 times DO ***
for c = 1 to 4
    rem *** Get next number ***
    Print("Enter number ")
    Sync()
    Sleep(1000)
    no = GetButtonEntry()
    rem *** If number smaller, record it ***
    if no < smallest
        smallest = no
    endif
next c
rem *** Display smallest value ***
PrintC("Smallest value entered was ")
Print(smallest)
Sync()
do
loop

```

Modified version of *Smallest*:

```

rem *** Find Largest Number Entered ***

rem *** Include Button functions ***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Get first number ***
Print("Enter number ")
Sync()
Sleep(2000)
no = GetButtonEntry()
rem *** Set largest to first number ***
largest = no
rem *** FOR 4 times DO ***
for c = 1 to 4

```

Activity 5.15

Modified version of *SumDice*:

```

rem *** Total dice throws ***

rem *** set total to zero ***
total = 0
rem *** for 5 times do ***
for c = 1 to 5
    rem *** Display roll number ***
    PrintC("Roll number ")
    Print(c)
    Sync()
    Sleep(1000)
    rem *** throw both dice ***
    dice1 = Random(1,6)
    dice2 = Random(1,6)
    rem *** display throw number and dice values ***
    PrintC("dice 1 : ")
    PrintC(dice1)
    PrintC("      dice 2 : ")
    Print(dice2)
    Sync()
    Sleep(2000)
    rem *** if either dice is a 1 then quit loop ***
    if dice1 = 1 and dice2 = 1
        exit
    endif
    rem *** add dice throws to total ***
    total = total + dice1 + dice2
next c
rem *** display final score ***
PrintC("Your final score was : ")
Print(total)
Sync()
do
loop

```

Activity 5.16

```

rem *** Display average of 6 scores ***

rem *** Include Button functions ***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Set total to zero ***
total = 0
rem *** FOR 6 times DO ***
for c = 1 to 6
    rem *** Get valid score ***
    Print("Enter score ")
    Sync()
    Sleep(2000)
    score = GetButtonEntry()
    while score < 0 or score > 100
        Print("Score must lie between 0 and 100")
        Print("Enter score ")
        Sync()
        Sleep(2000)
        score = GetButtonEntry()
    endwhile
    rem *** Add score to total ***
    total = total + score
next c
rem *** Calculate average ***
average = total/6
rem *** Display average ***

```

```
PrintC("Average = ")
Print(average)
Sync()
do
loop
```

Activity 5.17

No solution required.

Activity 5.18

The output would be:

```
-2 0
-2 1
-2 2
-2 3
-1 0
-1 1
-1 2
-1 3
0 0
0 1
0 2
0 3
1 0
1 1
1 2
1 3
```

On the computer screen, all output would occur on the same line with a slight delay between each set of values.

Activity 5.19

The code contains a `while` loop so we need to create three sets of test data to allow zero, one and more than one iteration of the loop.

Possible test values are:

	<i>num</i>	Expected Results (for <i>average</i>)
Test 1	0	0
Test 2	8,0	8
Test 3	12,6,0	9

Code for *Average*:

```
rem *** Calculate average of values entered ***

rem *** Include Button functions ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()

total = 0
count = 0
Print("Enter number (0 to stop)")
Sync()
Sleep(2000)
num = GetButtonEntry()
while num <> 0
    total = total + num
    count = count + 1
    Print("Enter number (0 to stop)")
    Sync()
    Sleep(2000)
    num = GetButtonEntry()
endwhile
average = total / count
PrintC("Average is ")
Print(average)
Sync()
do
loop
```

When we run the program with the test data, it turns out that all the results are as we expected.

However, this is more by good fortune than the fact that the code is foolproof.

The line

```
average = total/count
```

would, in most languages, cause the program to crash when we did the first test. This is because *count* would have the value zero and hence the calculation would cause a division by zero error. However, as we saw back in Chapter 3, AGK BASIC returns zero when division by zero is performed - just the answer we want!

However, you really should guard against this problem. For example, if you were to rewrite your code in C++, then that division by zero calculation would cause a crash.

We can solve the problem by changing the code to

```
if count = 0
    average = 0
else
    average = total / count
endif
```


6

Resources - A First Look

In this Chapter:

- ☐ Introducing Images
- ☐ Introducing Sprites
- ☐ Sound
- ☐ Music
- ☐ Introducing Text
- ☐ Introducing User Interaction

Resources - A First Look

Introduction

Any additional visual components or files that we make use of within an AGK project are known as **resources**. Typical resources are: images, sounds, music, sprites, buttons and even text.

Every resource is assigned an integer ID value. No two resources of the same type may have the same ID. However, resources of different types may share the same ID. So, it's okay for an image, say, to have an ID of 1 and a sound resource to also have an ID of 1.

A resource's ID can be chosen by the programmer or automatically by the program itself.

Any separate files required by a resource must be copied into the project's *media* folder.

Images

Image Formats

The type of image you create using your camera or download from the web is a **bitmap** image. A bitmap image is constructed from a series of coloured dots known as **pixels**. You have probably come across this term before, since the resolution of any screen or camera is usually quoted in pixels. For example, the Apple iPad 1 & 2 screen has a resolution of 768 pixels by 1024 pixels.

The more pixels an image contains, the more detail it will hold. Therefore, we often talk about the resolution of an image as being its size in pixels. Many cameras can easily obtain image resolutions of over 4000 by 3000 pixels.

The other simple way to create a bitmap image is to use a paint package such as Adobe Photoshop or even the modest Paint program included with Microsoft Windows.

Many painting packages can resize images. This allows you to shrink or expand the number of pixels in an image. Decreasing the size of an image means that some of the details that were in the original image will be lost. On the other hand, increasing an image's size cannot create detail that was not there in the original and can often make the enlarged image look fuzzy and slightly out of focus.

Image files can be stored in many formats. Some formats will save an exact copy of the original image (known as **lossless** formats) but others lose a small amount of the original's detail (**lossy** formats). This second option doesn't sound like a great idea, but the reason such formats are popular - in fact, the most widely used of all - is because these **lossy** formats use compression techniques to create much smaller files. A lossy image can be stored in a file that is only 10% or even 5% of the **lossless** file equivalent.

AGK BASIC recognises three image file formats. These are: BMP, PNG and JPG. BMP and PNG are lossless file formats and so should only be used for relatively small images; perhaps character figures and other visual components of a game. JPG

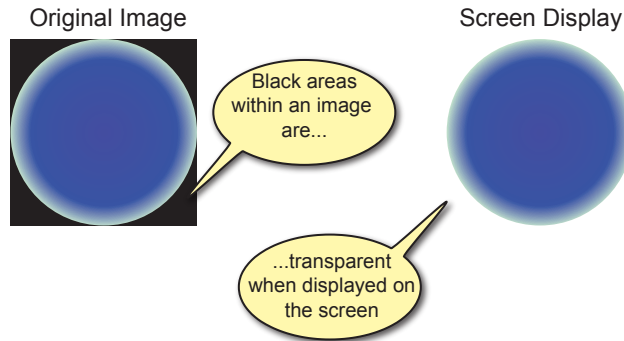
is a lossy format and is ideal for use with photographs and larger graphics. The degree of compression used when saving a file in JPG format can be specified. Less compression means a better quality image but a larger file.

Image Transparency

Images are always rectangular in shape. So how do you create a game that displays a football or a spaceship or anything else that isn't rectangular? All we need to do is make part of the image transparent. In AGK, there are two methods of achieving transparent areas within a displayed image. One option is to make black areas within an image invisible on the screen (see FIG-6.1).

FIG-6.1

Black Pixel
Transparency



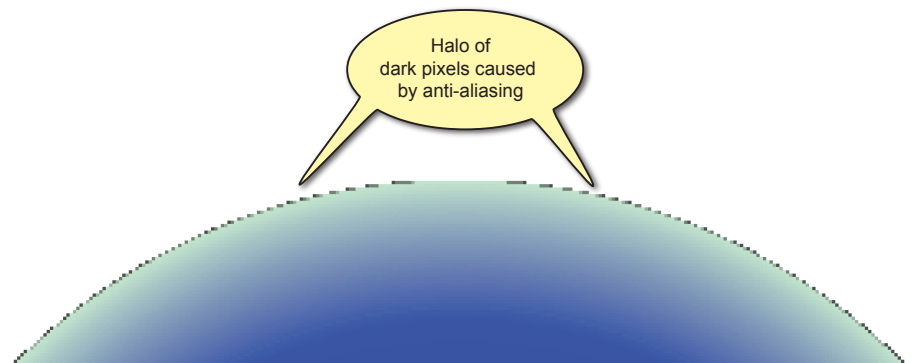
However, there are three things to be careful of when using this option:

- Only pixels which are truly black (red, green and blue intensities = 0) are made invisible. Part of the image which look black to you may not be completely black and therefore will not appear transparent when displayed.
- You have to make sure that no part of the image that should remain visible contains black pixels.
- A final, and perhaps more subtle problem, is caused by anti-aliasing.

Anti-aliasing is an attempt by image manipulation software to blend the edges of objects within an image in such a way as to give a smooth transition from one object to the next. This helps hide the pixelated nature of a digital image and in most cases improves the image. However, it can cause havoc when trying to create a transparent background. When anti-aliasing has been used in an image, the transition from visible area to the black invisible area will have a halo of near-black pixels and this halo will be all too visible when your image appears on screen (see FIG-6.2).

FIG-6.2

Anti-aliasing



To avoid the halo problem, make sure anti-aliasing is switched off when you are creating an image. Using black pixels to produce transparency does have its limitations. For example, it does not allow us to create semi-transparent elements within an image.

JPG files cannot have an alpha channel.

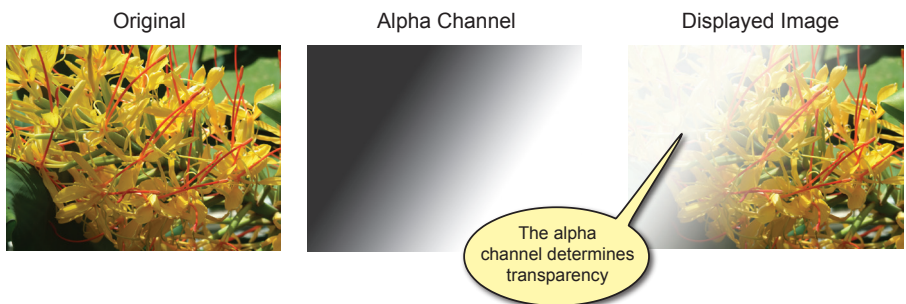
A second option for creating transparency is to include an **alpha channel** in the image itself.

We already know that an image is constructed from a sequence of pixels and that the colour of each pixel is determined by the intensity of its red, green and blue, components. These three colour components are sometimes referred to as the image's **colour channels**. Some image formats allow you to add a fourth channel known as the **alpha channel**. This channel is a grey-scaled overlay of the image surface and determines the transparency setting for every pixel within the image. In an area where the alpha channel is black, the image is fully transparent; where the alpha channel displays white, the image is opaque; and where the alpha channel is grey, the image is translucent. The shade of grey determines the degree of translucency.

FIG-6.3 shows an image, its alpha channel and how that image looks when displayed on screen.

FIG-6.3

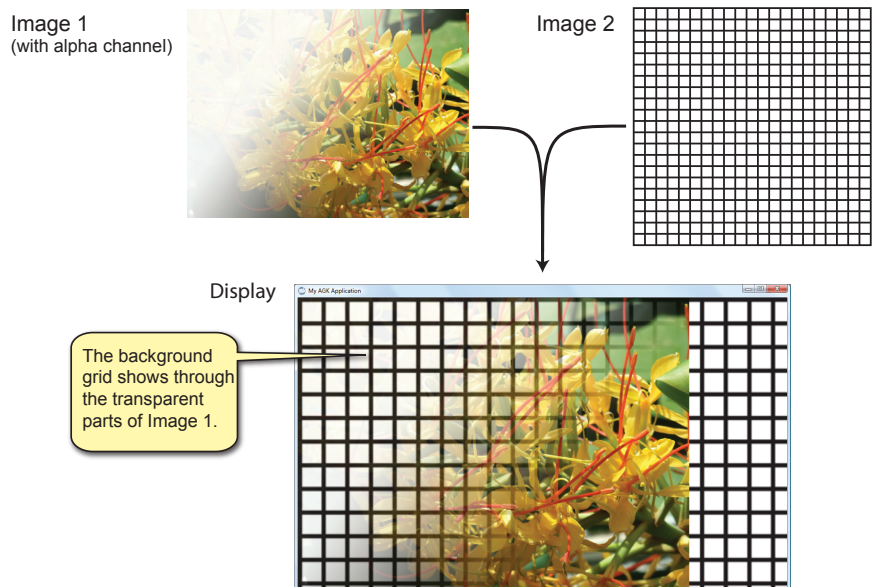
An Image with an Alpha Channel



The transparency is more obvious if we place a second image behind the original one (see FIG-6.4).

FIG-6.4

Alpha Channel Transparency



BMP and PNG files both allow alpha channel information to be stored (though in slightly different ways).

LoadImage()

If we want to display one or more images in a game, we need to start by copying the files containing the images into the AGK project's *media* folder. Next we need to issue a command to load each image into the game itself. This is done using the `LoadImage()` statement. There are two variations on this statement (see FIG-6.5).

FIG-6.5

LoadImage()

Version 1

`LoadImage (id , sfile [, iflag])`

Version 2

`integer LoadImage (sfile [, iflag])`

where:

- | | |
|--------------|---|
| id | is an integer value specifying the ID to be assigned to the image. This value must be 1 or above. No two images may have the same ID value. |
| sfile | is a string giving the name of the file containing the image. The file must be in the <i>media</i> folder for this project. |
| iflag | is an integer (0 or 1) which is used to determine how transparency is handled when the image is displayed. If <i>iflag</i> has the value zero, then the alpha channel of the image sets the transparency; if the value is 1, then the alpha channel is ignored and all black pixels within the image are made invisible. A value of zero is assumed if this parameter is omitted. |

Using the first version of this command, you need to specify the ID being assigned to the image for the duration of the program. For example, if the first image to be loaded is called "*ball.bmp*", then we would load the image using the statement

```
LoadImage(1,"ball.bmp",1)
```

This will assign the ID value of 1 to the image and black pixels will be invisible. Alternatively, we could use version 2 of the statement and write

```
id = LoadImage("ball.bmp",1)
```

This time the program decides on the ID to be assigned, but IDs are assigned in ascending order starting at 10001, so, as long as this is the first image to be loaded it will be assigned an ID of 10001.

Using the second version guarantees that we will not attempt to assign the same ID to two different images (which would, in any case, produce an error).

CreateSprite()

Although all images need to be loaded before they can be used, in order to see an image on the screen, you'll need to load that image into a sprite. To do this you need to create a sprite and specify the image to be displayed by the sprite. This is done using the `CreateSprite()` statement (see FIG-6.6).

FIG-6.6

CreateSprite()

Version 1

CreateSprite (id , imageId)

Version 2

integer CreateSprite (imageId)

where:

id is an integer value specifying the ID to be assigned to the sprite. This value must be 1 or above. No two sprites may have the same ID value.

imageId is an integer value specifying the ID of the image being copied into the sprite. This image must previously have been loaded using a `LoadImage()` statement. Use 0 to create a white sprite without an image.

At this stage we can think of a sprite as nothing more than an image which appears on the screen. But, as we will discover later, there are many sprite-related commands which allow us to do various operations such as move, rotate, resize and detect sprite collisions.

Like the two versions of `LoadImage()`, the two options in the `CreateSprite()` statement allow you to choose between deciding on the ID number yourself (version 1) or letting the program decide for you (version 2 - assigned values start at 10001).

In the example we are about to create, we will assign our own ID numbers since it uses only a single image and a single sprite. So, to create a sprite showing the ball image, we would first load the image and then create the sprite:

```
LoadImage(1,"ball.bmp",1)
CreateSprite(1,1)
```

Notice that the image and sprite have both been assigned an ID of 1. This is not a problem since they are two different types of objects (image and sprite). Only when you assign the same ID to two objects of the same type do you cause an error. Now we are ready to create a program to display our first image (see FIG-6.7).

FIG-6.7

Displaying a Sprite

When a sprite is first created, its top left corner is at position (0,0) - the top left corner of the app window.

```
rem *** First Sprite ***
rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
Sync()
do
loop
```

Activity 6.1

Create a new project called *FirstSprite*. Compile the default code in order to create the project's *media* folder. From the files you downloaded to accompany this book, go to the *AGKDownloads/Chapter 6* folder and copy the file *ball.bmp* to the project's *media* folder.

Change the contents of *main.agc* to match that given in FIG-6.7. Run and save the project. What is strange about the image?

AGK has a problem with sizing the image. Since we are working with a percentage-based screen layout, it has no idea exactly how large to make the sprite. It handles this by assuming the physical size of the image represents the percentage required. The ball image is 64 pixels wide by 64 pixels high, so AGK assumes you want the

image to take up 64% of the width and 64% of the height of the app window. Unfortunately, this is nowhere near the actual size we want.

SetSpriteSize()

The `SetSpriteSize()` statement allows use to specify the dimensions of a sprite. The sizes are given as a percentage of the screen, or in virtual pixels, depending on the option chosen when the program was created. The statement has the format shown in FIG-6.8.

FIG-6.8

`SetSpriteSize()`

`SetSpriteSize` (`id` , `fx` , `fy`)

where:

- | | |
|-----------|---|
| id | is the integer value previously assigned as the ID of the sprite to be resized. |
| fx | is a real value giving the width required. This value is given as a percentage of the screen width or in virtual pixels as appropriate. |
| fy | is a real value giving the height required (percentage or virtual pixels). |

So, if we wanted the ball sprite to occupy only 10% of the screen, we would use the line:

```
SetSpriteSize(1,10,10)
```

Activity 6.2

Modify *FirstSprite* by adding the `SetSpriteSize()` statement given above. Run the program and see how this changes the image displayed.

Change the height setting in *setup.agc* to 1024. Rerun the program. How is the sprite affected? Save your project.

As you can see from Activity 6.2, making the sprite 10% in both directions works only when the app window is square. Increasing the app window height also means an increase in the height of the sprite and our ball is no longer circular.

To solve this problem, `SetSpriteSize()` allows you to set the actual size of one dimension and use the value -1 for the other. When you choose this option, AGK works out the second dimension automatically to ensure that the sprite retains its original shape. For example, if we set the `fx` parameter to 10 and `fy` to -1 using the line

```
SetSpriteSize(1,10,-1)
```

the sprite will return to its round shape.

Of course, setting the `fy` to 10 and `fx` to -1 with

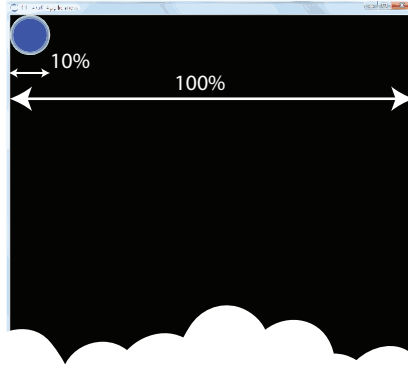
```
SetSpriteSize(1,-1,10)
```

will still result in a round ball, but it will be larger since 10% of the app window's height is much greater than 10% of its width (see FIG-6.9).

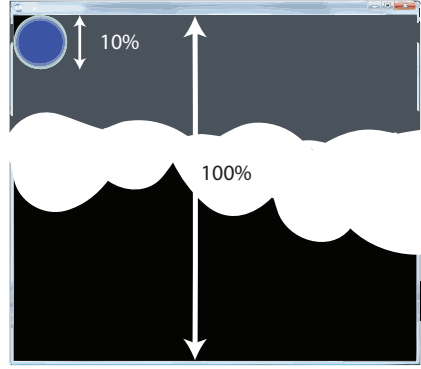
FIG-6.9

How Sprite Size Changes

SetSpriteSize(1,10,-1)



SetSpriteSize(1,-1,10)



Activity 6.3

Modify *FirstSprite* to use the -1 parameter in `SetSpriteSize()`. Try out both options, making the width -1 on the first run and the height -1 on the second run.

Save your project.

The only problem now with our sprite app is that, since the app window background is black, we really can't see if the black areas of the sprite are, indeed, invisible.

Activity 6.4

Add a `SetClearColor()` statement to your *FirstSprite* program to create a white background. (You'll also need to add an extra `Sync()` statement.)

Are the black pixels within the ball image invisible?

Save your project.

SetSpritePosition()

An existing sprite can be moved to a new position on the screen using the `SetSpritePosition()` statement which has the format shown in FIG-6.10.

FIG-6.10

SetSpritePosition()

SetSpritePosition ((id , fx , fy)

where:

- id** is the integer value previously assigned as the ID of the sprite to be moved.
- fx** is a real value giving the new x-coordinate (percentage or virtual pixels).
- fy** is a real value giving the new y-coordinate. Measured in virtual pixels or percentage.

By default, it is the top left corner of a sprite that is placed at the position specified.

Activity 6.5

In *FirstSprite*, add a two second delay and then move the sprite to the centre of the app window. Test and save your project.

By placing the `SetSpritePosition()` statement within a `for` loop and using the loop counter as a parameter, we can get the sprite to travel across the window.

Activity 6.6

Remove your last modification from *FirstSprite* and replace it with the following code:

```
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
    Sleep(50)
next p
```

Test the new version of the project.

SetSpriteDepth()

The program in FIG-6.11 is an extension of your *FirstSprite* project and demonstrates one sprite passing “behind” another.

FIG-6.11

Demonstrating Sprite
Depth

```
rem *** Sprite Depth ***
rem *** Clear screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load images ***
LoadImage(1,"ball.bmp",1)
LoadImage(2,"poppy.bmp",1)
rem *** Create ball sprite ***
CreateSprite(1,1)
SetSpriteSize(1,10,-1)
rem *** Create poppy sprite ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
    Sleep(50)
next p
do
loop
```

Activity 6.7

Modify your *FirstSprite* project to match the code given in FIG-6.11.

Test and save your project.

The ball passes “behind” the poppy because the ball sprite was created before the poppy. If we had wanted the ball to pass over the poppy, then we could have achieved this by having created the ball sprite after the poppy sprite. But another option is available; we can adjust the depth of a sprite using the `SetSpriteDepth()` statement. Sprite depth can be set to any value from 0 to 10000.

In original hand-drawn cartoons, the overall image is made up of a layer of transparent acetates. Different elements of the picture were drawn on different acetates. Those elements on the top-most acetate were at the “front” and those on the bottom acetate were at the “back”. AGK depth settings are equivalent to those acetate layers: depth 0 is at the “front”; depth 10000 is at the “back”.

The format of the `SetSpriteDepth()` statement is shown in FIG-6.12.

FIG-6.12

`SetSpriteDepth()`

`SetSpriteDepth` (`id` , `idepth`)

where:

id is the integer value previously assigned as the ID of the sprite.

idepth is an integer value giving the layer setting. A lower number will bring the sprite “forward” towards the top layer. This value can be in the range 0 to 10,000.

When a sprite is created, it is assigned a default layer of 10. Sprites on the same layer have a depth determined by the order in which they were created (as we have already seen).

Activity 6.8

Modify *FirstSprite*, assigning the ball sprite to layer 9 immediately after its creation. How does this affect the program’s display? Save your project.

GetSpriteDepth()

To determine the current depth of a sprite, use the `GetSpriteDepth()` statement (see FIG-6.13).

FIG-6.13

`GetSpriteDepth()`

integer `GetSpriteDepth` (`id`)

where:

id is the integer value previously assigned as the ID of the sprite.

CloneSprite()

You can make a copy of a sprite using the `CloneSprite()` statement. This will make an exact copy of the sprite specified. The statement’s format is shown in FIG-6.14.

FIG-6.14

`CloneSprite()`

`CloneSprite` (`id` , `idToCopy`)

where:

id is the integer value of the ID to be assigned to the new sprite.

idToCopy is an integer value giving the ID of the existing sprite to be cloned.

Whatever characteristics have been set for the original sprite (size, transparency, depth, etc.) will be duplicated in the clone.

Activity 6.9

Modify *FirstSprite*, making a copy of the poppy sprite and positioning it at (20,20).

Assign the new sprite a depth setting of 8. What happens as the ball passes the two poppies? Save your project.

SetSpriteVisible()

We can make a sprite invisible - and make it reappear - using the `SetSpriteVisible()` statement which has the format shown in FIG-6.15.

FIG-6.15

SetSpriteVisible()

`SetSpriteVisible (id , invisible)`

where:

id is the integer value previously assigned as the ID of the sprite.

invisible is an integer value (0 or 1) specifying that the sprite is to be hidden (0) or made visible (1).

Activity 6.10

Modify *FirstSprite* so that the two poppy sprites are hidden after the ball has moved to the bottom of the screen. Save your project.

DeleteSprite()

When a sprite is no longer required by a program, that sprite can be deleted. Although deletion is not necessary, it does free up resources on the machine which can, in turn, speed up your game. Sprites are deleted using the `DeleteSprite()` statement whose format is shown in FIG-6.16.

FIG-6.16

DeleteSprite()

`DeleteSprite (id)`

where:

id is an integer value giving the ID of the sprite to be deleted.

DeleteAllSprites()

If your program contains several sprites, they can all be deleted, using the `DeleteAllSprites()` statement (see FIG-6.17).

FIG-6.17

DeleteAllSprites()

`DeleteAllSprites ()`

DeleteImage()

When an image is no longer required by a sprite, or when the sprite using an image has been deleted, then that image can be deleted, thereby freeing up further resources. To delete an image we use the `DeleteImage()` statement (see FIG-6.18).

FIG-6.18

DeleteImage()

`DeleteImage (id)`

where:

id is an integer value giving the ID of the image to be deleted.

Deleting a resource only deletes it from the computer's memory; the actual file containing the resource is not affected.

DeleteAllImages()

Rather than delete images individually, you can delete every loaded image using the `DeleteAllImages()` statement (see FIG-6.19).

FIG-6.19

DeleteAllImages()

`DeleteAllImages ()`

Of course, you should only call this statement when every image in the program is no longer being used by other program elements such as a sprite.

There are many more sprite commands and these will be covered in later chapters.

Sound

Sound files, like image files, come in many different formats. And like those for images, some formats are lossy, but have small file sizes, while others are lossless with larger file sizes. The current version of AGK will handle only uncompressed WAV sound files.

To play a sound, the file containing that sound must first be copied into the project's *media* folder. Within the program we can then load and play the file.

LoadSound()

Like images, sounds must be loaded before they can be used. This is done using the `LoadSound()` statement (see FIG-6.20).

FIG-6.20

LoadSound()

Version 1

`LoadSound (id , sfile)`

Version 2

`integer LoadSound (sfile)`

where:

id is an integer value specifying the ID to be assigned to the sound file.

sfile is a string giving the name of the file to be loaded. This must be a WAV file and must be stored in the project's *media* folder.

In the first version of the statement the program chooses the ID number; in the second version the ID value is automatically selected by AGK and returned by the statement.

PlaySound()

Once loaded, a sound file can be played using the `PlaySound()` statement (see FIG-6.21).

FIG-6.21

PlaySound()

`PlaySound (id [, ivol [, iloop [, iprrty]]])`

where:

- | | |
|---------------|---|
| id | is an integer value specifying the ID previously assigned to the sound. |
| ivol | is an integer value (0 to 100) representing the volume setting. The default setting is 100. |
| iloop | is an integer value (0 or 1) which determines if the sound is to play continuously. If set to 0, the sound will play only once; if set to 1, the sound will be repeated. Zero is the default value. |
| iprrty | is an integer value which is designed to be used to set the sound's priority. This option is currently not implemented. |

Several sound files can
be played at the same
time.

StopSound()

When a sound is set to play only once, it will, obviously, stop when the end of the file is reached, but if you want playing to stop prematurely, you can do so using the `StopSound()` statement. This statement has the format shown in FIG-6.22.

FIG-6.22

StopSound()

`StopSound (id)`

where:

- | | |
|-----------|---|
| id | is an integer value giving the ID of the sound that is to be stopped. |
|-----------|---|

DeleteSound()

When a sound resource is no longer required, it is best to delete that resource from your program. This can be done using the `DeleteSound()` statement (see FIG-6.23 for format).

FIG-6.23

DeleteSound()

`DeleteSound (id)`

where:

- | | |
|-----------|---|
| id | is an integer value giving the ID of the sound that is to be deleted. |
|-----------|---|

SetSoundSystemVolume()

Although the volume of a specific sound is set when that sound is first loaded and cannot be adjusted later, the system volume can be adjusted at any time using the `SetSoundSystemVolume()` statement which has the format shown in FIG-6.24.

FIG-6.24

SetSoundSystemVolume()

The diagram shows the function name "SetSoundSystemVolume" in a rounded rectangle, followed by an opening parenthesis "(", a green box containing the parameter "ivol", and a closing parenthesis ")", all enclosed in a larger rounded rectangle.

where:

ivol is an integer (0 to 100) giving the percentage volume adjustment. For example, 50 would give half volume, 100 would leave the volume unchanged.

GetSoundExists()

You can check that a sound with a specific ID value currently exists using `GetSoundExists()` (see FIG-6.25).

FIG-6.25

GetSoundExists()

integer
 The diagram shows the function name "GetSoundExists" in a rounded rectangle, followed by an opening parenthesis "(", a green box containing the parameter "id", and a closing parenthesis ")", all enclosed in a larger rounded rectangle.

where:

id is an integer value giving the ID of the sound to be checked.

The statement will return 1 if a sound of the specified ID currently exists; otherwise zero is returned.

GetSoundsPlaying()

We can also check the number of instances of a sound that are playing at the same time. `GetSoundsPlaying()` returns the number of instances of a specified sound currently in existence (see FIG-6.26).

FIG-6.26

GetSoundsPlaying()

integer
 The diagram shows the function name "GetSoundsPlaying" in a rounded rectangle, followed by an opening parenthesis "(", a green box containing the parameter "id", and a closing parenthesis ")", all enclosed in a larger rounded rectangle.

where:

id is an integer value giving the ID of the sound whose number of instances is to be returned.

GetSoundInstances()

The `GetSoundInstances()` statement performs exactly the same purpose as `GetSoundsPlaying()` and so the two statements are interchangeable. The statement's syntax is shown in FIG-6.27.

FIG-6.27

GetSoundInstances()

integer
 The diagram shows the function name "GetSoundInstances" in a rounded rectangle, followed by an opening parenthesis "(", a green box containing the parameter "id", and a closing parenthesis ")", all enclosed in a larger rounded rectangle.

where:

id is an integer value giving the ID of the sound whose number of instances is to be returned.

Activity 6.11

Start a new project called *Sounds*. Compile the default code to create the *media* folder. Copy the file *J1to10.wav* from the *AGKDownloads/Chapter6* to the project's *media* folder.



Activity 6.11 (continued)

Recode the contents of *main.agc* to read:

```
LoadSound(1,"J1to10.wav")
PlaySound(1)
do
loop
```

Make sure the sound is activated and the volume turned up on your computer.

Compile and run the program. Does the sound play? Save your project.

When the program plays a sound file it does not halt execution of the other statements in your program while the sound is played. It merely passes the sound file details to your sound card, leaves the sound card to deal with playing the file, and then gets on with executing the other statements in your program.

Activity 6.12

Modify the code in *Sounds* so that it displays the numbers 1 to 10 as the sound file plays. The code for this is:

```
LoadSound(1,"J1to10.wav")
PlaySound(1)
for c = 1 to 10
    Print(c)
    Sync()
    Sleep(1000)
next c
do
loop
```

Test the program. Does the sound stop when the `Sleep(1000)` statement is executed? Save your project.

We have seen in previous chapters that the `Sleep()` statement halts the program for a specified time. However, since the sound file is being handled by the sound card, any sounds already being played are not affected by the `Sleep()` statement.

Activity 6.13

In this Activity we are going to examine what is required in order to have a sound file played repeatedly.

Remove the `for..next` loop and its loop body from *Sounds*.

Change the line

```
PlaySound(1)
```

to

```
PlaySound(1,100,1)
```

so that the sound should play repeatedly at full volume. Run the program. Does the sound play more than once?

Inside the `do..loop` add the line

```
Sync()
```

How does this affect the playing of the sound file? Save your project.

So the `Sync()` statement needs to be executed in order for the sound to play continuously. This is because the `Sync()` statement does more than just update the screen. It handles details about other things within the program including making sure sound files are replayed when appropriate.

Music

Music files are handled separately from sound files and although some of the commands for handling music look very similar to those for sounds, there are major differences.

AGK currently plays only MP3, OGG Vorbis and ACC formatted music files.

LoadMusic()

The `LoadMusic()` statement loads a specified music file and assigns it an ID number. The statement has the format shown in FIG-6.28.

FIG-6.28

LoadMusic()

Version 1

LoadMusic ((id , sfile))

Version 2

integer LoadMusic (sfile)

where:

- id** is an integer value specifying the ID to be assigned to the music file.
- sfile** is a string giving the name of the file to be loaded, This must be an MP3, OGG Vorbis or AAC file and must be stored in the project's *media* folder.

Automatically assigned ID values start at 1.

In the first version of the statement, the programmer chooses the ID number; in the second version, the ID value is automatically selected by AGK and returned by the statement.

PlayMusic()

Once loaded, a music file is played using the `PlayMusic()` statement (see FIG-6.29).

FIG-6.29

PlayMusic()

PlayMusic (([id [, iloop [, idStrt [, idFin]]]]))

where:

- id** is an integer value giving the ID of the music file to be played.
- iloop** is an integer value (0 or 1) which determines if the music is to play continuously. If set to 0, the music will play only once; if set to 1, the music will be repeated. Zero is the default value.
- idStrt** is an integer value giving the lowest ID of the list of music files to be played.

Only one music file can be playing at any one time.

idFin is an integer value giving the highest ID of the list of music files to be played.

This command will play all or most of the MP3 files stored in the *media* folder without explicitly specifying all the ID numbers. To stop this you need to use the longest form of the command and state explicitly which file or group of files are to be played.

The simplest version of this command is

```
PlayMusic()
```

which will play the music file with the lowest ID. For example, if a program started with the lines

```
LoadMusic(1,"TrackA.mp3")
LoadMusic(2,"TrackB.mp3")
LoadMusic(3,"TrackC.mp3")
LoadMusic(4,"TrackD.mp3")
LoadMusic(5,"TrackE.mp3")
```

These tracks would have to be stored in the project's *media* folder.

and followed this with

```
PlayMusic()
```

then *TrackA* would be played first and then all other tracks played in sequence.

```
PlayMusic(2,0)
```

would play *TrackB* followed by *TrackC*, *TrackD* and *TrackE*. The tracks would be played once only.

```
PlayMusic(3,1)
```

would play *TrackC*, *TrackD*, and *TrackE* and then play all five tracks continuously.

```
PlayMusic(1,1,3,5)
```

would play *TrackA*, *TrackB* then repeat *TrackC*, *TrackD* and *TrackE* continuously.

```
PlayMusic(3,0,3,3)
```

would play *TrackC* once only.

Using this command also requires you to add a **Sync()** statement within the **do...loop** structure.

Activity 6.14

For copyright reasons, no MP3 files are included in the downloads for this book.

Start a new project called *Music*. Compile the default code to create the project's *media* folder. Copy three of your own MP3 files into the *media* folder.

Modify *main.agc* to load all three files but play only the last one. The file should be played only once. Test and save your code.

PauseMusic()

You can pause a playing MP3 file using the `PauseMusic()` statement. This has the format shown in FIG-6.30.

FIG-6.30

PauseMusic()

PauseMusic ()

Note that there is no need for an ID parameter since only one music file can be playing at any instant.

ResumeMusic()

A paused MP3 file can be resumed from the point where it paused using the `ResumeMusic()` statement (see FIG-6.31).

FIG-6.31

ResumeMusic()

ResumeMusic ()

StopMusic()

To stop a music file completely use `StopMusic()` (see FIG-6.32).

FIG-6.32

StopMusic()

StopMusic ()

DeleteMusic()

When a music resource is no longer required you can use the `DeleteMusic()` statement to free up the memory occupied by the file (see FIG-6.33).

FIG-6.33

DeleteMusic()

DeleteMusic (id)

where:

id is an integer value giving the ID of the music resource to be deleted from the program.

We can determine various characteristics about music files from several other music statements.

GetMusicExists()

The `GetMusicExists()` statement returns 1 if a music resource of a specified ID currently exists; otherwise zero is returned (see FIG-6.34).

FIG-6.34

GetMusicExists()

integer GetMusicExists (id)

where:

id is an integer value giving the ID of the music resource to be checked.

SetMusicFileVolume()

You can set the volume of a specific music file using the `SetMusicFileVolume()` (see FIG-6.35).

FIG-6.35

SetMusicFileVolume()

SetMusicFileVolume (id , ivol)

where:

- id** is an integer value giving the ID of the music whose volume is to be changed.
- ivol** is an integer giving the volume as a percentage of full volume (0 - silent; 100 - full volume).

SetMusicSystemVolume()

To set the volume for every music track, the `SetMusicSystemVolume()` statement can be used (see FIG-6.36).

FIG-6.36

SetMusicSystemVolume()

`SetMusicSystemVolume ((ivol))`

where:

- ivol** is an integer giving the volume as a percentage of full volume (0 - silent; 100 - full volume).

Detecting User Interaction

Most programs react to the user clicking a mouse or touching a pressure-sensitive screen. AGK uses three main commands to detect a mouse/screen press.

GetPointerPressed()

One of these commands is the `GetPointerPressed()` statement which has the format shown in FIG-6.37.

FIG-6.37

GetPointerPressed()

integer `GetPointerPressed ()`

The statement returns 1 immediately the press occurs. Before and after that instant, zero is returned.

GetPointerReleased()

A complementary statement is `GetPointerReleased()` which returns 1 the instant the mouse button is released, or the finger lifted from the screen. This statement has the format shown in FIG-6.38.

FIG-6.38

GetPointerReleased()

integer `GetPointerReleased ()`

GetPointerState()

This third statement returns 1 while the button or finger is being pressed down and returns 0 when the button/finger is not pressed. Note this is different from the first two statements which only return 1 for a single instant as the mouse/finger is pressed/lifted. The `GetPointerState()` command has the format shown in FIG-6.39.

FIG-6.39

GetPointerState()

integer `GetPointerState ()`

The code in FIG-6.40 demonstrates the use of the `GetPointerPressed()` and `GetPointerReleased()` statements.

FIG-6.40

Using Pointer
Statements

```

Sync()
do
    rem *** Check for press ***
    if GetPointerPressed()=1
        Print("Pressed")
    endif
    rem *** Check for release ***
    if GetPointerReleased()=1
        Print("Released")
    endif
    Sync()
loop

```

Notice that for the first time, the main code is within the `do..loop` structure which loops continually while testing for the button/screen press.

Activity 6.15

Start a new project called, *PressedFlower* and change the code in *main.agc* to match that given in FIG-6.40.

Test the program and check that you can see messages as you press and release the mouse button. Save your project.

If we are not interested in detecting the exact moment the button is pressed or released, but want to know if the button/finger is currently pressed down/touching the screen or up/not touching the screen, then the `GetPointerState()` command will be more useful.

Activity 6.16

Modify the code in *PressedFlower* to read:

```

Sync()
do
    if GetPointerState()=1
        Print("Pressed")
    else
        Print("Released")
    endif
    Sync()
loop

```

Test the new code. How do the messages that appear on the screen differ from those displayed by the previous version of the program? Save your project.

GetPointerX() and GetPointerY()

We can find out the exact position on the screen where a press has occurred using `GetPointerX()` (which returns the x-coordinate) and `GetPointerY()` (which returns the y-coordinate). The formats for these two statements are shown in FIG-6.41.

FIG-6.41

GetPointerX()
GetPointerY()

integer `GetPointerX ()`

integer `GetPointerY ()`

Activity 6.17

Modify the code in *PressedFlower* by removing the line

```
Print("Pressed")
and replacing it with
PrintC(GetPointerX())
PrintC(" ")
Print(GetPointerY())
```

Test and save your project.

GetSpriteHit()

We can find out if a particular screen position is over a sprite using the `GetSpriteHit()` command. This is useful for finding out if the user has, for example, clicked/pressed on a sprite. The command's format is shown in FIG-6.42.

FIG-6.42

GetSpriteHit()

integer `GetSpriteHit` ((fx , fy)

where:

fx, fy are real numbers giving the position within the app window to be tested. The values will represent percentages or virtual coordinates depending on the window setup.

If the location is over a sprite, the sprite ID is returned, otherwise zero is returned.

Activity 6.18

Modify *PressedFlower* by removing all of the code within the `do..loop` structure.

Add code to display a sprite showing *poppy.bmp* at the centre of the app window (set the sprite's width to 15%).

To hide the poppy when it is clicked on, change the code within the `do..loop` structure to:

```
if GetPointerPressed()=1
  x# = GetPointerX()
  y# = GetPointerY()
  hit = GetSpriteHit(x#,y#)
  if hit <> 0
    SetSpriteVisible(1,0)
  endif
endif
Sync()
```

Test and save your project.

Text Resources

We've already seen how to display information on the screen using the `Print()` statement, but its main limitation is that we cannot choose the exact position at which the output is to appear. This will be a critical requirement for any game.

► Text resources use the same character images as `Print()` to form the displayed text.

Luckily, AGK offers a second and more controlled way of creating textual output; **text resources**. Just like images, sprites, sound, and music resources, text resources are created and assigned a unique ID.

A few of the many statements available for manipulating text resources are described here.

CreateText()

The `CreateText()` statement allows us to create a new text resource. The statement has the format shown in FIG-6.43.

FIG-6.43

CreateText()

Version 1

CreateText (id , string)

Version 2

integer CreateText (string)

where:

id is an integer value specifying the ID to be assigned to the text resource.

string is a string containing the text to be held within the text resource.

Version 1 of the statement allows the programmer to select the resource ID; version 2 automatically assigns an ID and returns that ID.

For example, we could create a text resource containing the phrase *Hello world*, assigning it an ID of 1 using the statement:

```
CreateText(1, "Hello world")
```

SetTextColor()

FIG-6.44

SetTextColor()

We can select the color and transparency of the text using the `SetTextColor()` statement (see FIG-6.44).

SetTextColor (id , ired , igreen , iblue , itrans)

where:

id is an integer value specifying the ID of the text resource whose colour is to be set.

ired is an integer value specifying the intensity of the red component of the colour. Range 0 to 255.

igreen is an integer value specifying the intensity of the green component of the colour. Range 0 to 255.

iblue is an integer value specifying the intensity of the blue component of the colour. Range 0 to 255.

The default colour for a text resource is white.

itrans is an integer value specifying the opacity of the text. Range 0 (invisible) to 255 (fully opaque).

For example, if we have already created a text resource with an ID of 1, then we can display that text in opaque black using the line:

```
SetTextColor(1,0,0,0,255)
```

SetTextPosition()

By default, text will appear in the top left corner of the app window. To position it elsewhere we need to use the `SetTextPosition()` statement which has the format shown in FIG-6.45).

FIG-6.45

SetTextPosition()

SetTextPosition ((id , x , y))

where:

id is the integer value previously assigned as the ID of the text to be moved.

x is a real value giving the new x-coordinate. This will be in virtual pixels or percentage depending on the coordinate system defined when the app window was created.

y is a real value giving the new y-coordinate measured in virtual pixels or percentage.

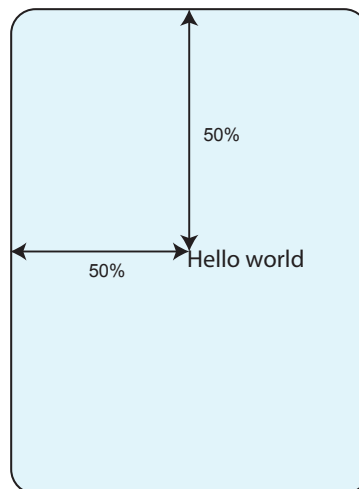
We could place text resource 1 at the centre of the app window using the statement:

```
SetTextPosition(1,50,50)
```

The position (50,50) refers to the top left part of the text (see FIG-6.46).

FIG-6.46

Positioning a Text Resource



SetTextSize()

The size of the text can be adjusted using the `SetTextSize()` statement (see FIG-6.47).

FIG-6.47

SetTextSize()

SetTextSize ((id , fsize))

where:

id	is the integer value previously assigned as the ID of the text to be resized.
fsize	is a real value specifying the height of the characters within the text. This is measured in percentage or virtual pixels depending on the setup. The width is calculated automatically.

The default size for all text output is 4. Remember also that the larger the text becomes, the more obvious the limitations of the images from which it is derived.

We could change the size of the text displayed by text resource 1 from the default 4 to 6 using the statement:

```
SetTextSize(1,6)
```

SetTextString()

The actual text contained within a text resource can be changed using the `SetTextString()` statement (see FIG-6.48).

FIG-6.48

SetTextString()

`SetTextString` ((`id` , `string`)

where:

id	is the integer value previously assigned as the ID of the text resource whose text is to be changed.
string	is the new string to be assigned to the text resource.

SetTextVisible()

You can hide a text resource or make it reappear using the `SetTextVisible()` statement (see FIG-6.49).

FIG-6.49

SetTextVisible()

`SetTextVisible` ((`id` , `ivisible`)

where:

id	is the integer value previously assigned as the ID of the text resource to be operated on.
ivisible	is an integer value (0 or 1) used to hide or display the text. (0 - hide ; 1 - show)

DeleteText()

When a text resource is no longer required, it should be deleted, thereby freeing up memory resources. This is done using the `DeleteText()` statement (see FIG-6.50).

FIG-6.50

DeleteText()

`DeleteText` ((`id`)

where:

id is an integer value giving the ID of the text resource to be deleted from the program.

DeleteAllText()

If your program contains several text resources and you wish to remove all of them, use `DeleteAllText()` (see FIG-6.51).

FIG-6.51

DeleteAllText()

DeleteAllText ()

Using a Text Resource

The program below demonstrates most of the text resource statements we have covered here. The purpose of the code is to display a sequence of dots. Starting with one dot and increasing to 10 before starting again at one dot. This sequence is repeated five times before the program stops. A simple animation such as this might be used to indicate to the user that the program is busy.

The program's logic can be described in structured English as:

```
Create empty text resource
Set text colour
Set text size
Set text position
FOR 5 times DO
    Create empty string
    FOR dots = 1 TO 10 DO
        Add dot to string
        Place string in text resource
        Wait 200 msecs
    ENDFOR
    Empty text resource
    Wait 1 sec
ENDFOR
Delete text resource
```

The code for the program is shown in FIG-6.52.

FIG-6.52

Using a Text Resource

```
rem *** Text Resource demo ***

rem *** Create empty string ***
CreateText(1,"")
rem *** Set resource attributes ***
SetTextPosition(1,15,30)
SetTextColor(1,250,250,0,255)
SetTextSize(1,10)
rem *** FOR 5 times DO ***
for c = 1 to 5
    rem *** Empty string ***
    text$ = ""
    for dots = 1 to 10
        rem *** Add dot to string ***
        text$ = text$+"."
        rem *** Place string in text resource ***
        SetTextString(1,text$)
        Sync()
        rem *** Wait 200 msecs ***
        Sleep(200)
    next dots
```



FIG-6.52

(continued)

Using a Text Resource

```
rem *** Empty text resource ***
SetTextString(1,"")
Sync()
rem *** Wait one second ***
Sleep(1000)
next c
rem *** Delete resource ***
DeleteText(1)
Sync()
do
loop
```

Activity 6.19

Start a new project called *UsingText* and modify the code in *main.agc* to match that given in FIG-6.52. Test the program.

Modify the code to use the underscore character (`_`) instead of the full stop.

Test and save your project.

Later

This chapter has covered all of the statements available for manipulating sound and music resources. However, there are many other commands that can be used with images, sprites, text and user input which are not covered here. These will be explained in later chapters.

Summary

- Resources is the name given to other elements added to a project. These can be images, sounds, music, sprites, virtual buttons, or text.
- A resource needs to be created and assigned an ID before it can be used.
- No two resources of the same type may be assigned the same ID number.
- Resources of different types may have identical ID numbers.
- As a general rule, resources should be deleted when no longer required.
- Files containing resources must be stored in the project's *media* folder.
- Most images are constructed from colour dots known as pixels.
- An image constructed from pixels is known as a bitmap image.
- Bitmap images can be stored in many different formats.
- Lossless formats save an exact copy of an image but create large files.
- Lossy formats save a degraded copy of the image but create smaller files.
- AGK can handle three bitmap formats: BMP, PNG, and JPG.
- BMP and PNG are lossless file formats; JPG is a lossy file format.
- Images can contain transparent elements.
- Transparency can be achieved in one of two ways: by making all black pixels

invisible or by adding an alpha channel to the image.

- Alpha channels allow degrees of translucency.
- When creating an image in which black elements are to be made invisible make sure that the image has not been created using anti-aliasing.
- Anti-aliasing can cause problems around the edges of objects within an image.
- Images need to be loaded into AGK and given a unique ID number.
- To display an image on the screen it must first be loaded into a sprite.
- Using the default setup, screen distances are given in percentage terms and sprites use the pixel size of the image it contains as a percentage value when determining the size of the image.
- Sprites can be resized, moved, and made invisible.
- Sprites can be placed on different layers.
- There are 10,001 layers numbered 0 to 10,000.
- Layer 0 is the top layer; layer 10,000 is the bottom layer.
- A sprite placed on a higher layer will pass in front of a sprite placed on a lower layer.
- A sprite can be cloned.
- A sprite can be made invisible.
- Deleting a sprite frees up the resources it requires.
- Sound files must be in uncompressed WAV format.
- A sound can be set to play one time only or repeatedly.
- The volume of an individual sound can be set only when playing starts.
- The overall system volume can be modified at any time.
- Music files must be in MP3 OGG Vorbis or AAC formats.
- By default, all music files are played once when a `PlayMusic()` command is issued.
- Basic user interaction allows us to detect a screen touch or mouse button press.
- It is possible to detect when:
 - the mouse button/screen is first pressed
 - the mouse button/screen is first released
 - the current state of the mouse button/screen - pressed or unpressed.
- We can detect if a mouse/screen press occurs over a sprite.
- Using a text resource allows us to control attributes of a string.
- The string within a text resource can be modified, resized, positioned, coloured, and made transparent.

Solutions

Activity 6.1

Although the image is only 64 x 64 pixels it appears much larger within the app window.

Activity 6.2

Modified *FirstSprite*:

```
rem *** First Sprite ***

rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Resize sprite ***
SetSpriteSize(1,10,10)
Sync()
do
loop
```

The sprite now occupies 10% of the width and height of the app window. Because the app window is square, this means that the ball is perfectly round.

To modify the app window height, the *height* line in *setup.agc* needs to be changed to

```
height=1024
```

When the height of the app window is changed, 10% of the height is much greater than 10% of the width and so the ball becomes stretched.

Activity 6.3

The line

```
SetSpriteSize(1,10,10)
```

should first be changed to

```
SetSpriteSize(1,-1,10)
```

The ball will be round but this time it is 10% of the height and so, much larger than previously.

On the next run the line should now read

```
SetSpriteSize(1,10,-1)
```

which will return the ball to the size it had been before we resized the app window (10% of the width).

Activity 6.4

Modified *FirstSprite*:

```
rem *** First Sprite ***

rem *** Clear screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Resize sprite ***
SetSpriteSize(1,10,-1)
Sync()
do
loop
```

The black pixels are invisible.

Activity 6.5

Modified *FirstSprite*:

```
rem *** First Sprite ***

rem *** Clear screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Resize sprite ***
SetSpriteSize(1,10,-1)
Sync()
rem *** Wait then reposition sprite ***
Sleep(2000)
SetSpritePosition(1,50,50)
Sync()
do
loop
```

Activity 6.6

Modified *FirstSprite*:

```
rem *** First Sprite ***

rem *** Clear screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Resize sprite ***
SetSpriteSize(1,10,-1)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
  SetSpritePosition(1,p,p)
  Sync()
  Sleep(50)
next p
do
loop
```

Activity 6.7

No solution required.

Activity 6.8

Modified *FirstSprite*:

```
rem *** Sprite Depth ***

rem *** Change screen to white ***
SetClearColor(255,255,255)
Sync()

rem *** Load images ***
LoadImage(1,"ball.bmp",1)
LoadImage(2,"poppy.bmp",1)
rem *** Create ball sprite ***
CreateSprite(1,1)
rem *** Bring sprite forward ***
SetSpriteDepth(1,9)
SetSpriteSize(1,10,-1)
rem *** Create poppy sprite ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
  SetSpritePosition(1,p,p)
  Sync()
  Sleep(50)
next p
do
loop
```

The ball passes in front of the poppy rather than behind it.

Activity 6.9

Modified *FirstSprite*:

```

rem *** Sprite Depth ***

rem *** Change screen to white ***
SetClearColor(255,255,255)
Sync()

rem *** Load images ***
LoadImage(1,"ball.bmp",1)
LoadImage(2,"poppy.bmp",1)
rem *** Create ball sprite ***
CreateSprite(1,1)
rem *** Bring sprite forward ***
SetSpriteDepth(1,9)
SetSpriteSize(1,10,-1)
rem *** Create poppy sprites ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
CloneSprite(3,2)
SetSpritePosition(2,20,20)
rem *** Move cloned sprite to layer 8 ***
SetSpriteDepth(3,8)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
    Sleep(50)
next p
do
loop

```

The ball passes under the new poppy and over the original poppy.

Activity 6.10

Modified *FirstSprite*:

```

rem *** Sprite Hide ***

rem *** Change screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load images ***
LoadImage(1,"ball.bmp",1)
LoadImage(2,"poppy.bmp",1)
rem *** Create ball sprite ***
CreateSprite(1,1)
rem *** Bring sprite forward ***
SetSpriteDepth(1,9)
SetSpriteSize(1,10,-1)
rem *** Create poppy sprites ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
CloneSprite(3,2)
SetSpritePosition(2,20,20)
rem *** Move cloned sprite to layer 8 ***
SetSpriteDepth(3,8)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
    Sleep(50)
next p
rem *** Hide poppies ***
SetSpriteVisible(2,0)
SetSpriteVisible(3,0)
Sync()
do
loop

```

Activity 6.11

The sound file *J1to10.wav* should play if everything is set up properly.

The sound file voices the numbers 1 to 10 in Japanese.

Activity 6.12

The text should be in sync with the spoken words. Although the speaker pauses, the sound plays continuously even while the `sleep()` statement is being executed.

Activity 6.13

Modified *Sounds*:

```

rem *** Play sound file ***
rem *** Load file ***
LoadSound(1,"J1to10.wav")
rem *** Start playing file ***
PlaySound(1,100,1)
do
    Sync()
loop

```

Without the `Sync()` statement the file will play only once.

Activity 6.14

Code for *Music*:

```

rem *** Play music ***

rem *** Load music Files ***
LoadMusic(1,"TrackA.mp3")
LoadMusic(2,"TrackB.mp3")
LoadMusic(3,"TrackC.mp3")
rem ** Play last track once ***
PlayMusic(3,0,3,3)
do
loop

```

Activity 6.15

The messages will appear briefly as the mouse button is pressed and released.

Activity 6.16

The *Pressed* message remains visible while the mouse button is down; the *Released* message remains visible while the mouse button is up.

Activity 6.17

Modified *PressedFlower*:

```

Sync()
do
    if GetPointerState()=1
        PrintC(GetPointerX())
        PrintC(" ")
        Print(GetPointerY())
    else
        Print("Released")
    endif
    Sync()
loop

```

Activity 6.18

Modified *PressedFlower*:

```

rem *** Load image ***
LoadImage(1,"poppy.bmp")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpritePosition(1,50,50)
SetSpriteSize(1,15,-1)
Sync()
do
    rem *** IF pointer pressed THEN ***
    if GetPointerPressed()=1
        rem *** Get its coordinates ***
        x# = GetPointerX()
        y# = GetPointerY()
        rem *** Check if coord over a sprite ***
        hit = GetSpriteHit(x#,y#)
        rem ***IF they are THEN hide sprite ***
        if hit <> 0
            SetSpriteVisible(1,0)
        endif
    endif
    Sync()
loop

```

Activity 6.19

Modified *UsingText*:

```
rem *** Text Resources demo ***

rem *** Create empty string ***
CreateText(1,"")
rem *** Set resource attributes ***
SetTextPosition(1,15,30)
SetTextColor(1,250,250,0,255)
SetTextSize(1,10)
rem *** FOR 5 times DO ***
for c = 1 to 5
  rem *** Empty string ***
  text$ = ""
  for dots = 1 to 10
    rem *** Add underscore to string ***
    text$ = text$+"_"
    rem *** Place string in text resource ***
    SetTextString(1,text$)

    Sync()
    rem *** Wait 200 msecs ***
    Sleep(200)
  next dots
  rem *** Empty text resource ***
  SetTextString(1,"")
  Sync()
  rem *** Wait one second ***
  Sleep(1000)
next c
rem *** Delete resource ***
DeleteText(1)
Sync()
do
loop
```

7

Spot the Difference Game

In this Chapter:

- ☐ Designing Screen Layouts
- ☐ Creating Sprite Images
- ☐ Adding Background Music
- ☐ Adding Sound Effects
- ☐ Changing Screen Orientation
- ☐ Game Testing

Game - Spot the Difference

Introduction

At last, we know enough AGK BASIC to create a first game. This game is a 21st century update on the spot-the-difference game so beloved of many magazines. The game shows two almost identical images and the challenge is to spot the differences between the two images.

Game Design

When creating a game, there are many aspects of that game that we have to think about before we start to write program code.

Since this is a computer game derived from an existing paper-based one, we don't have to worry about giving an in-depth description of the game, defining the rules or stating how the game is won.

On the other hand, we still need to design the screen layout for the game. In fact, there may be several layouts to design: a start-up splash screen, the main game screen, an end-game screen and a credits screen detailing all those involved in the game development. Not only the overall screen designs need to be considered, but also the design of any individual sprites that may appear during the game play.

Any background music and sound effects not only have to be created, but when these are to be played also needs to be specified.

User interaction methods and help options are other aspects that have to be considered.

Game Description

In our game, the player is presented with two almost identical images. The left-hand image is the original image; the right-hand image has six modifications. The aim of the game is for the player to click (press) on the areas of the right-hand image that differ from those in the left-hand image.

The time elapsed since the start of the game is continually displayed.

The total time (in seconds) taken to find all six differences is displayed at the end of the game.

Screen Layouts

This game will have four screen layouts: splash screen, game screen, finish screen and credits screen.

You may want to create a rough drawing of the various screen layouts before going on to create a more detailed design using a drawing or paint package.

Another important point at this stage is to consider the screen size and resolution of the device(s) on which you want the game to run. Although AGK will allow your game to run on almost any platform, you may still want to consider how the screen size will affect the playability of your game. For example, 10 buttons along the right-hand edge of an iPad looks fine, but try the same thing on an iPhone and only the

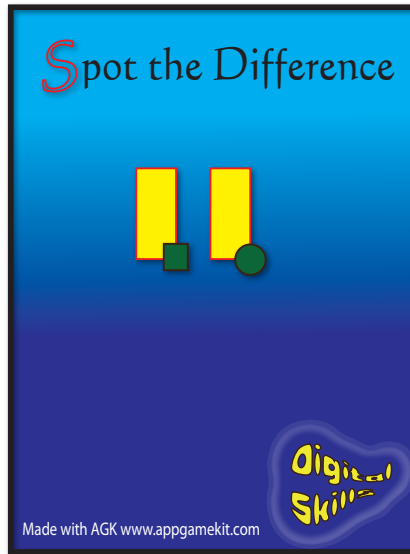
smallest of fingers will be able to use the buttons easily! And what about the near future? If you create images which are 1024 x 768 pixels in size with the iPad 2 in mind, what happens if a later iPad has a screen resolution of 2048 x 1536 pixels? Your images may not look as good on that!

For this game, the screen layouts have been designed using Adobe Illustrator which is a vector-drawing package. The great advantage of a vector-based image is that it can be converted to a regular bitmap image giving the best possible quality for a required resolution.

The splash screen (filename : *AGK.Splash.png*) is shown in FIG-7.1.

FIG-7.1

The Splash Screen



This is a single PNG image. Note that it includes the name of the game, the company name (Digital Skills), text stating that it was built using AGK and the AGK website address. This last element is requested of you by **The Game Creators** if you are going to publish your app, but is not compulsory.

The second image (see FIG-7.2) is of the game screen containing the two photographs that form the game. This is the only image in landscape mode.

FIG-7.2

The Main Screen

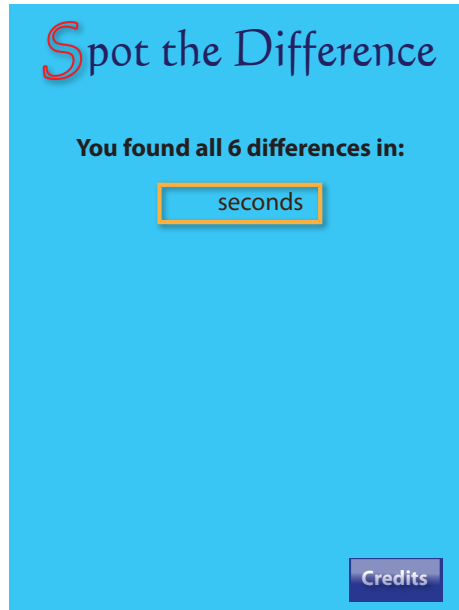


The photos themselves are not separate entities but part of the single overall image. Note that the top right corner leaves a gap where the time is to be displayed in real-time.

The third image is the end screen which shows the total time taken in seconds (see FIG-7.3).

FIG-7.3

The End Screen

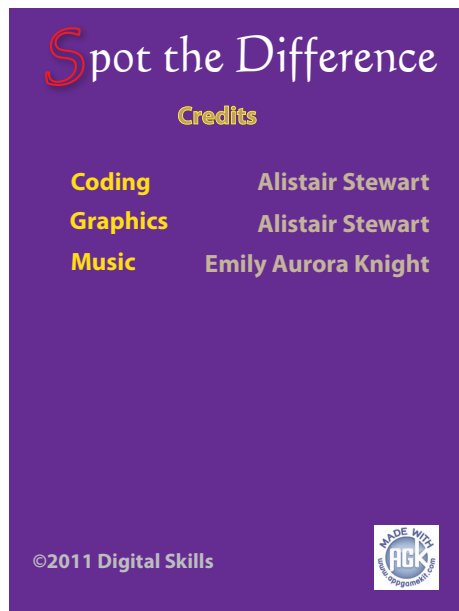


Again, you can see that a space has been left for the actual number of seconds taken to find all the differences. In addition, this screen also shows a separate button sprite in the bottom-right which allows the user to view the credits screen if required.

The final screen (see FIG-7.4) shows the names of those involved in creating the various aspects of the game: graphics, code, music. It also adds copyright details and the AGK logo.

FIG-7.4

The Credits Screen



A final visual component is the ring which appears around the differences in the photograph when the player presses in the correct area. Although there will be six of these, all make use of the same image (see FIG-7.5).

FIG-7.5

The Circle Spite



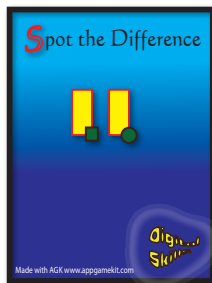
Other Resources

The only other resources used in the game are a sound effect, which plays when a modified area of the photo is pressed for the first time, and music which plays in the background while the game is running.

Overall Game Document

FIG-7.6

The Overall Game Document

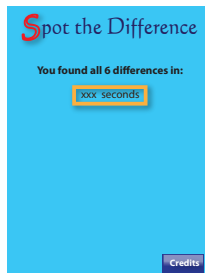


Splash Screen



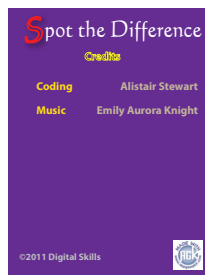
Main Screen

- Music begins
- Rings appear around correctly selected areas
- Sound effect when ring appears
- Time in seconds displayed



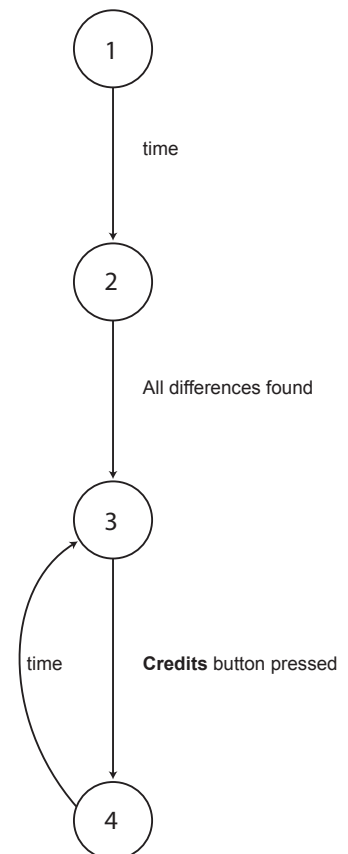
End Screen

- Music continues
- Displays total time taken to find all differences



Credits Screen

- Music continues



In the *Main* and *End* screen layouts X's are used to indicate where text is to be positioned, but the exact value of that text is unknown at the time of the design.

The *Main* screen is in landscape mode, while the other three screens are designed for portrait mode. As a general rule, it is best not to switch between modes within a game, but in this example it is interesting to see how the actual game play experience is affected by the transition.

On the right of FIG-7.6 is a **state-transition** diagram. The numbered circles represent the four different screen layouts. When each new screen appears during the game we consider the game to have entered a new **state**. The lines between the circles represent the moving from one state to another (i.e. from one screen to another). The text beside the lines explains what causes the game to move from one state to another. So we see that we move from the splash screen to the main game screen once an unspecified amount of time has passed; we move from the main screen to the end screen when all 6 differences have been found. Notice that we move to the credits screen only if the **Credits** button is pressed and that we return from the credits screen to the end screen after some time has elapsed.

For a more complex game, we might need to give greater detail for the design of each screen and the individual sprites which may appear on that screen.

Copyright Issues

Of course, if you intend to create a game simply for the amusement of yourself and your family, then making use of images you find on the internet, or adding your favourite music to the game isn't really a problem. However, should you wish to turn your game into a commercial product then you must make sure all aspects of the game are either copyright free, that you have permission from the copyright holder to use the material, or that the material is entirely of your own creation.

Even if you created the photographs used in a game, you can still breach copyright. For example, you can't use someone's image in a commercial product without their approval. You can't even use some buildings! If you were to use images taken in a Disney park for example, you would probably have their lawyers on your doorstep before you had made your first 10 sales!

Even if you record your own music, the melody itself may be copyrighted. Play and write your own music to be on the safe side.

You mustn't even borrow a one second sound effect without approval.

Don't worry! There are websites which offer copyright free material - but check that it can be used in a commercial product.

Finally, the images have no copyright problems, you have written and played the music, created all the sound effects, so you must be safe now, right? Afraid not! If you save your music in MP3 format, you'll find another set of lawyers wanting to have a few words. This time it won't happen until you've sold 5000 copies of your game but at that point you'll have to hand over large sums of money for the privilege of using the MP3 format. The way round this one is to use the OGG Vorbis format for your music files. AGK will automatically look for a file in this format even when your code specifies MP3.

And once you've made sure all your resources have no copyright issues, are you safe at last to write your game? Well, not entirely. You can still be on the receiving end of

a legal communication if someone thinks you've ripped off their game idea or even if your code makes use of some technique that has been copyrighted.

Have you given up all hope of creating a commercial game? Well, you can do a few things to protect yourself from the unexpected legal challenge. One option is to set up a limited company and publish your games through that (it's really not too complicated). Using this method, only your company can be sued if the worst should happen - not you. So you won't have to sell your home and flash new car to pay all the legal claims that have arrived on the doorstep.

And perhaps the easiest option of all is to let The Game Creators publish your game for you. Okay they are going to want 30%, but on the other hand they will test your game, suggest any changes, market it for you, even add revenue-gathering adverts and organise the cut-down free version and the paid-for full version. Chances are you'll sell more copies through them than you would do on your own and even after giving them their cut, you'll still make more money. And perhaps best of all, they are legally responsible - not you. Now, on with the game ...

Game Logic

The next stage is to do a high-level structured English description of the game.

The first level should be kept short:

- 1 Load resources
- 2 Set up game screen
- 3 Play game
- 4 End game

More detail can be added to each of these using stepwise refinement:

- 1 Load resources
 - 1.1 Load images
 - 1.2 Load sound
 - 1.3 Load music
- 2 Set up game screen
 - 2.1 Start music
 - 2.2 Display Main screen
 - 2.3 Add circles over differences
 - 2.4 Hide circles
- 3 Play game
 - 3.1 Start timer
 - 3.2 REPEAT
 - 3.3 IF user selected a difference THEN
 - 3.4 Show ring
 - 3.5 Play sound effect
 - 3.6 ENDIF
 - 3.7 Update time
 - 3.8 UNTIL all 6 differences selected
 - 3.9 Delete Main screen resources
- 4 End game
 - 4.1 Show End screen
 - 4.2 Display time taken
 - 4.3 Display Credits button
 - 4.4 DO
 - 4.5 IF Credits button pressed THEN
 - 4.6 Show Credits screen for 5 seconds
 - 4.7 ENDIF
 - 4.8 LOOP

Game Code

The game code follows the logic given above. The first section loads the resources but also includes comments on the overall program.

Structured English:

Load resources

Code:

```
rem *****
rem * program      : Spot the Difference *
rem * version      : 1.0                  *
rem * language     : AGK BASIC v1.02      *
rem * date         : 18 Aug 2011          *
rem * author       : A. Stewart           *
rem * platform     : Ipad 1               *
rem *****

rem *** Load resources ***

rem *** Load images ***
main = LoadImage("Main.jpg")
finish = LoadImage("End.jpg")
credits = LoadImage("Credits.jpg")
ring = LoadImage("Ring.png",0)
button = LoadImage("Button.bmp",1)

rem *** Load sounds ***
ringsound = LoadSound("Click.wav")

rem *** Load music ***
backgroundmusic = LoadMusic("Background.wav")
```

Structured English:

Set up game screen

Code:

```
rem *** Play music ***
PlayMusic(BackgroundMusic)

rem *** Show main screen ***
CreateSprite(1,main)
SetSpriteSize(1,100,100)

rem *** Load rings at image differences ***
CreateSprite(2,ring)
SetSpriteSize(2,-1,10)
SetSpritePosition(2,91,86)
CloneSprite(3,2)
SetSpritePosition(3,51.5,22)
CloneSprite(4,2)
SetSpritePosition(4,49,68)
CloneSprite(5,2)
SetSpritePosition(5,73,66)
CloneSprite(6,2)
SetSpritePosition(6,88.5,66)
CloneSprite(7,2)
```

```

SetSpritePosition(7,55.75,62.5)

rem *** Hide rings ***
for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
next c
rem *** Update screen ***
Sync()

```

Structured English:

Play game

Code:

```

rem *** Start timer ***
start = GetSeconds()
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,255)
SetTextPosition(1,88,6)

rem *** Set count of differences found ***
found = 0
rem *** Get user clicks until all 6 differences found ***
repeat
    rem *** Check for clicked button ***
    pressed = GetPointerPressed()
    rem *** IF pressed, then check for sprite hit ***
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF clicked over hidden ring THEN
        if hit > 1 and hit <=7 and GetSpriteVisible(hit)=0
            rem *** Show ring ***
            SetSpriteVisible(hit,1)
            rem *** Play sound effect ***
            PlaySound(1)
            rem *** Add 1 to differences found ***
            found = found + 1
        endif
    endif
    rem *** Update time so far ***
    timetaken = GetSeconds() - start
    SetTextString(1,Str(timetaken))
    Sync()
until found = 6

rem *** Delete existing sprites ***
for c = 1 to 7
    DeleteSprite(c)
next c
rem *** Delete sound ***
DeleteSound(ringsound)
rem *** Delete text ***
DeleteText(1)

```

Note that we have had to add a *found* variable to keep count of how many differences have been found.

Structured English:

End game

Code:

```
rem *** Show End screen... ***
CreateSprite(1,finish)
SetSpriteSize(1,100,100)
rem *** ... with button... ***
CreateSprite(2,button)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,80,90)
rem *** ...and total time taken ***
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,0,255)
SetTextPosition(1,36,31)
Sync()

rem *** Allow for Credits button press ***
do
    pressed = GetPointerPressed()
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF Credit button pressed THEN ***
        if hit = 2
            rem *** Show credits screen for 5 seconds ***
            CreateSprite(3,credits)
            SetSpriteSize(3,100,100)
            SetSpriteDepth(3,8)
            Sync()
            Sleep(5000)
            rem *** Remove Credits screen ***
            DeleteSprite(3)
        endif
    endif
    Sync()
loop
```

The *Credits* screen is displayed “on top of” the *End* screen, so when it is deleted after 5 seconds, the *End* screen reappears.

Activity 7.1

Start a new project called *SpotTheDifference* and compile the default code so that the project’s *media* folder is created.

From *AGKDownloads/Chapter7*, copy all the files in the folder to the project’s *media* folder.

In *setup.agc* set width to 1024 and height to 768. This will create a landscape oriented app window.

Modify *main.agc* to match the code given over the last few pages. Test and save your code. What problems occurred?

No program is likely to be perfect on the first attempt. Perhaps there will be problems with the code: the logic may be wrong and this will be highlighted during testing.

The main problem with this first version of the game is caused by the fact that the main screen is in landscape mode but the *End* and *Credits* screens are in portrait mode. To get this to operate correctly, we need to change the screen orientation after the game is complete.

SetDisplayAspect()

We can change the screen's aspect ratio using the `SetDisplayAspect()` statement. In this statement we set the ratio of the width to the height. At the start of a program, the aspect ratio is determined by the values given for *width* and *height* in the *setup.agc* file. When the program is running, we can change to portrait orientation (but without changing the actual app window dimensions) using the line:

One of the numbers has to be real so that the calculation will produce a real (not integer) result.

```
SetDisplayAspect(768/1024.0)
```

The `SetDisplayAspect()` statement has the format shown in FIG-7.7.

FIG-7.7

`SetDisplayAspect()`

`SetDisplayAspect` ((`value`))

where:

value is a real number giving the ratio of the width to the height.

Activity 7.2

Modify your program so that, immediately after the resources of the main screen have been deleted, the display ratio is set using the lines:

```
rem *** Reset aspect ratio ***
SetDisplayAspect(768.0/1024.0)
```

Retest and save your program.

An important aspect to check is the finer details of game playability. For example, you may have noticed that when the last difference is found, the game jumps immediately to the *End* screen without giving the player a chance to see the placing of that final ring. We could solve this problem by getting the program to pause for one second before the *End* screen appears.

Activity 7.3

Add the lines

```
rem *** Wait before showing next screen ***
Sleep(1000)
```

immediately after the `DeleteText(1)` line.

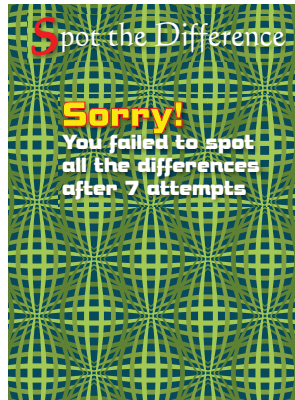
Test this modification and check that the player has time to see the final ring in position before the *End* screen appears.

A major problem with the game is that it has no way of stopping the player just pressing anywhere at random in the hope of hitting on a difference merely by chance. To stop this, we could introduce a maximum number of presses on the modified image. Perhaps 7 - this would allow the player one wrong attempt. However, introducing this change would mean that a new screen would have to be introduced

into the game, showing that the player had failed to complete the game. The *Failed* image is shown in FIG-7.8. This page will also show the **Credits** button.

FIG-7.8

The Fail Screen



This modification to the program means that various parts of the game documentation also need to be changed. The first of these is the overall game document showing the various pages of the game and the state-transition diagram. The updated version of this document is shown in FIG-7.9.

FIG-7.9

The Updated Game Document



Splash Screen



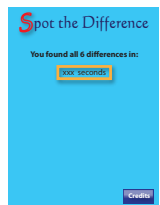
Main Screen

- Music begins
- Rings appear around correctly selected areas
- Sound effect when ring appears
- Time in seconds displayed



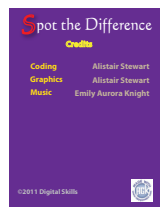
Failed Screen

- Music continues
- Displays failed message



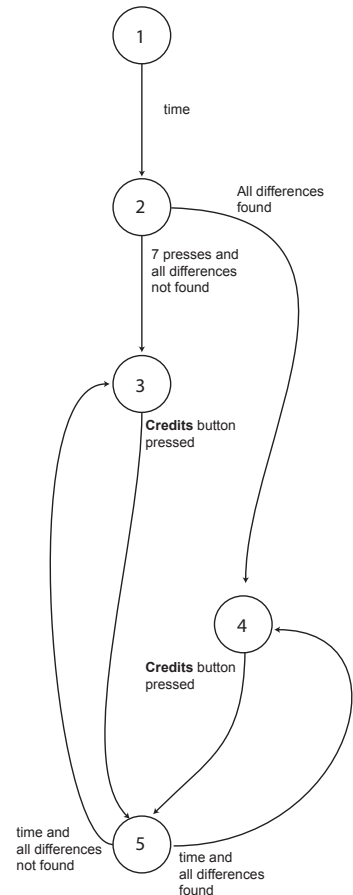
End Screen

- Music continues
- Displays total time taken to find all differences



Credits Screen

- Music continues



Note how much the state-transition diagram has changed. Not only have the state numbers assigned to the *End* and *Credits* screens changed, but the paths through the structure have become much more complex. From the *Main* screen (2) we may go to the *End* screen (4) if all 6 differences are found, but there is also an option to go to the *Fail* screen (3) when 7 presses have been made without all 6 differences being found. Both screens 3 and 4 have an option to show the *Credits* screen (5) for a set time period before screen 3 or 4 reappears. When the paths through the game start to become complex (as in this case), the state-transition diagram is a great way of maintaining an easy-to-follow overview of the whole game process.

The next part of the documentation to be changed is the structured English. Level 1 remains unchanged but the breakdown of some of its steps need to be modified. The updated logic is shown below with the changes highlighted.

```

3 Play game
  3.1 Start timer
  3.2 REPEAT
  3.3     IF user selected a difference THEN
  3.4         Show ring
  3.5         Play sound effect
  3.6     ENDIF
  3.7     Update time
  3.8 UNTIL all 6 differences selected or 7 presses made
  3.9 Delete Main screen resources

4 End game
  4.1 IF all 6 differences found THEN
  4.2     Show End screen
  4.3     Display time taken
  4.4 ELSE
  4.5     Show Fail screen
  4.6 ENDIF
  4.7 Display Credits button
  4.8 DO
  4.9     IF credits button pressed THEN
  4.10         Show Credits screen for 5 seconds
  4.11     ENDIF
  4.12 LOOP
  
```

Luckily, returning from the *Credits* screen to either the *End* or *Failed* screen isn't a problem since the *Credits* screen is shown on top of the previous screen. When the *Credits* screen is removed the appropriate screen will reappear.

Activity 7.4

Update your project to implement the changes described above. This requires the following steps:

- Copy the file *Fail.jpg* to the *media* folder.
- Add a line of code to load the image.
- The ID given to the image should be stored in the variable *fail*.
- Before the `repeat..until` loop, create a variable called *presscount* and set it to zero. Increment this variable every time `pressed = 1` is true.
- Add `or presscount = 7` to the condition in the `until` statement.
- Add the code for the new `if` statement described in the *End Game* structured English.

Check that the updated version of the program operates correctly by first winning a game and then losing one. Check that the *Credits* screen shows correctly in both cases. Resave your project.

Update the program's comments as appropriate.

Solutions

Activity 7.1

The *media* folder should contain the following files:

AGKSplash.png
Background.wav
Button.bmp
Click.wav
Credits.jpg
End.jpg
Main.jpg
Ring.png

The dimension setting lines in *setup.agc* should be changed to:

```
width=1024
height=768
```

The complete program code in *main.agc* is:

```
rem *****
rem * program      : Spot the Difference *
rem * version      : 1.0                 *
rem * language     : AGK BASIC v1.02     *
rem * date         : 18 Aug 2011         *
rem * author       : A. Stewart          *
rem * platform     : Ipad 1              *
rem *****

rem *** Load resources ***

rem *** Load images ***
main = LoadImage("Main.jpg")
finish = LoadImage("End.jpg")
credits = LoadImage("Credits.jpg")
ring = LoadImage("Ring.png",0)
button = LoadImage("Button.bmp",1)

rem *** Load sounds ***
ringsound = LoadSound("Click.wav")

rem *** Load music ***
backgroundmusic = LoadMusic("Background.wav")

rem *** Play music ***
PlayMusic(backgroundmusic)

rem *** Show main screen ***
CreateSprite(1,main)
SetSpriteSize(1,100,100)

rem *** Load rings at image differences ***
CreateSprite(2,ring)
SetSpriteSize(2,-1,10)
SetSpritePosition(2,91,86)
CloneSprite(3,2)
SetSpritePosition(3,51.5,22)
CloneSprite(4,2)
SetSpritePosition(4,49,68)
CloneSprite(5,2)
SetSpritePosition(5,73,66)
CloneSprite(6,2)
SetSpritePosition(6,88.5,66)
CloneSprite(7,2)
SetSpritePosition(7,55.75,62.5)

rem *** Hide rings ***
for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
next c
rem *** Update screen ***
Sync()

rem *** Start timer ***
start = GetSeconds()
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,255)
SetTextPosition(1,88,6)

rem *** Set count of differences found ***
found = 0
```

```
rem *** Get user clicks until all 6 differences
found ***
repeat
    rem *** Check for clicked button ***
    pressed = GetPointerPressed()
    rem *** IF pressed, check for sprite hit ***
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF clicked over hidden ring THEN
        if hit > 1 and hit <=7 and
            GetSpriteVisible(hit)=0
            rem *** Show ring ***
            SetSpriteVisible(hit,1)
            rem *** Play sound effect ***
            PlaySound(1)
            rem *** Add 1 to differences found ***
            found = found + 1
        endif
    endif
    rem *** Update time so far ***
    timetaken = GetSeconds() - start
    SetTextString(1,Str(timetaken))
    Sync()
until found = 6

rem *** Delete existing sprites ***
for c = 1 to 7
    DeleteSprite(c)
next c
rem *** Delete sound ***
DeleteSound(ringsound)
rem *** Delete text ***
DeleteText(1)

rem *** Show End screen... ***
CreateSprite(1,finish)
SetSpriteSize(1,100,100)
rem *** ... with button... ***
CreateSprite(2,button)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,80,90)
rem *** ...and total time taken ***
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,255)
SetTextPosition(1,36,31)
Sync()

rem *** Allow for Credits button press ***
do
    pressed = GetPointerPressed()
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF Credit button pressed THEN ***
        if hit = 2
            rem *** Show credits for 5 secs ***
            CreateSprite(3,credits)
            SetSpriteSize(3,100,100)
            SetSpriteDepth(3,8)
            Sync()
            Sleep(5000)
            rem *** Remove Credits screen ***
            DeleteSprite(3)
        endif
    endif
    Sync()
loop
```

The main problem is that although the main screen appears correctly, the *End* and *Fail* screens are not positioned correctly.

Activity 7.2

The new lines (shown in bold) should be placed as follows:

```
rem *** Reset aspect ratio ***
SetDisplayAspect(768.0/1024.0)

rem *** Show End screen... ***
CreateSprite(1,finish)
SetSpriteSize(1,100,100)
```

This modification should ensure the *End* screen is correctly sized.

Activity 7.3

The new lines (shown in bold) should be placed as follows:

```
DeleteText(1)

rem *** Wait before showing next screen ***
Sleep(1000)

rem *** Show End screen ***
CreateSprite(1,finish)
```

This gives a slight delay before the main screen disappears.

Activity 7.4

The file *Fail.jpg* should be added to the project's *media* file.

The final program code should be:

```
rem *****
rem * program      : Spot the Difference *
rem * version      : 1.1                *
rem * language     : AGK BASIC v1.02    *
rem * date         : 18 Aug 2011        *
rem * author       : A. Stewart         *
rem * platform     : Ipad 1             *
rem *****

rem *** Load resources ***

rem *** Load images ***
main = LoadImage("Main.jpg")
finish = LoadImage("End.jpg")
credits = LoadImage("Credits.jpg")
ring = LoadImage("Ring.png",0)
button = LoadImage("Button.bmp",1)
fail = LoadImage("Fail.jpg")

rem *** Load sounds ***
ringsound = LoadSound("Click.wav")

rem *** Load music ***
backgroundmusic = LoadMusic("Background.wav")

rem *** Play music ***
PlayMusic(BackgroundMusic)

rem *** Show main screen ***
CreateSprite(1,main)
SetSpriteSize(1,100,100)

rem *** Load rings at image differences ***
CreateSprite(2,ring)
SetSpriteSize(2,-1,10)
SetSpritePosition(2,91,86)
CloneSprite(3,2)
SetSpritePosition(3,51.5,22)
CloneSprite(4,2)
SetSpritePosition(4,49,68)
CloneSprite(5,2)
SetSpritePosition(5,73,66)
CloneSprite(6,2)
SetSpritePosition(6,88.5,66)
CloneSprite(7,2)
SetSpritePosition(7,55.75,62.5)

rem *** Hide rings ***
for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
next c
rem *** Update screen ***
Sync()

rem *** Start timer ***
start = GetSeconds()
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,255)
SetTextPosition(1,88,6)

rem *** Set count of differences found ***
found = 0
rem *** Number of clicks so far is zero ***
presscount = 0
rem *** Get user clicks until all 6 differences
found ***
repeat
    rem *** Check for clicked(pressed)
    pressed = GetPointerPressed()
```

```
rem *** IF pressed, ***
if pressed = 1
    rem *** Add 1 to clicks ***
    inc presscount
    rem *** Check for sprite hit ***
    x = GetPointerX()
    y = GetPointerY()
    hit = GetSpriteHit(x,y)
    rem *** IF clicked over hidden ring THEN
    if hit > 1 and hit <=7 and
        GetSpriteVisible(hit)=0
        rem *** Show ring ***
        SetSpriteVisible(hit,1)
        rem *** Play sound effect ***
        PlaySound(1)
        rem *** Add 1 to differences found ***
        found = found + 1
    endif
endif
rem *** Update time so far ***
timetaken = GetSeconds() - start
SetTextString(1,Str(timetaken))
Sync()
until found = 6 or presscount = 7

rem *** Delete existing sprites ***
for c = 1 to 7
    DeleteSprite(c)
next c
rem *** Delete sound ***
DeleteSound(ringsound)
rem *** Delete text ***
DeleteText(1)
rem *** Wait before showing next screen ***
Sleep(1000)
rem *** Reset aspect ratio ***
SetDisplayAspect(768.0/1024.0)
rem *** IF all differences found ***
if found = 6
    rem *** Show End screen... ***
    CreateSprite(1,finish)
    SetSpriteSize(1,100,100)
    rem *** ..and total time taken ***
    CreateText(1,Str(timetaken))
    SetTextColor(1,0,0,255)
    SetTextPosition(1,36,31)
else
    rem *** Show Fail screen... ***
    CreateSprite(1,fail)
    SetSpriteSize(1,100,100)
endif
Sync()
rem *** ... with button... ***
CreateSprite(2,button)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,80,90)

rem *** Allow for Credits button press ***
do
    pressed = GetPointerPressed()
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF Credit button pressed THEN ***
        if hit = 2
            rem *** Show credits for 5 secs ***
            CreateSprite(3,credits)
            SetSpriteSize(3,100,100)
            SetSpriteDepth(3,8)
            Sync()
            Sleep(5000)
            rem *** Remove Credits screen ***
            DeleteSprite(3)
        endif
    endif
    Sync()
loop
```


User-Defined Functions

In this Chapter:

- ☐ Creating Functions
- ☐ **gosub** and **return** Statements
- ☐ Mini-Specs
- ☐ Modular Programming Concepts
- ☐ Parameter Passing
- ☐ Return Values
- ☐ Pre-Conditions

Functions

Introduction

Look at the computer in front of you. Notice how it is made up of several separate components: keyboard, screen, mouse, and inside the main casing are other discreet pieces such as the hard disk and CDROM.

Why are computers made this way, as a collection of separate pieces rather than having everything encased in a single frame?

Well, there are several reasons. Firstly, by using separate components, each can be designed to perform just one specific task such as: get information from the user (the keyboard); display information (the screen); store information (the disk) etc. This allows all of these items to be made and tested separately.

Also, if a component breaks down or needs to be replaced, you simply have to unplug that component and replace it with a new one.

Why is all of this relevant to creating games programs? Years of experience have shown that the advantages of this modular approach to construction doesn't just apply to physical items such as computers, it also applies to software.

Rather than create a program which consists of one continuous set of instructions, we can split the program into several **routines** (also known as **modules**, **functions** or **subroutines**). Each routine is designed to perform just one specific function. This approach is particularly important in long programs and when several programmers are involved in creating the software.

In fact, routines in AGK BASIC are usually referred to as **functions**, and that's the term we'll use from here on in.

Functions

Designing a Function

The first stage in creating a function is to decide what task the function has to perform. For example, we might want a function to do something as simple as display a line of asterisks or move a sprite about the screen.

A good function will perform only a single task and be relatively short - perhaps no more than 20 to 30 lines of code (often much less).

When a team of people is involved in creating the software, it is important that the exact purpose of each function is written out in detail so there can be no misconceptions between the people designing the routine and those programming it.

Functions must also be given a name. This name should reflect the purpose of the function and often starts with a verb, since functions perform tasks.

So let's have a first attempt at writing out a design for a function that is to draw a line of asterisks.

FUNCTION NAME	: DrawLine
DESCRIPTION	: Draws a horizontal line of 10 asterisks.

This function document is known as a **mini-spec** and, although it does not yet show all the features that will appear in a full mini-spec, it contains all the details we need to create our simple function. The only tricky part is to write a description that is unambiguous - something that is not always that easy! Notice the word *horizontal* has been included so that there is no possibility of the programmer deciding to create a function that produces a vertical line of asterisks.

Activity 8.1

List any other details that might be added to the description to make the requirements more exact.

Coding a Function

From the mini-spec we get the name and purpose of the function. From that we can create the following code:

```
function DrawLine()  
    Print("*****")  
endfunction
```

Notice that the module begins with the keyword `function` and ends with the keyword `endfunction`.

The first line also contains the name of the function, *DrawLine*, and an empty set of parentheses.

Between the first and last lines go the set of instructions that perform the task the function has been designed to do. In this case, only one line of code is needed.

Calling a Function

The code within a function will only be executed if that function is **called**. To call a function, a program need only specify the function's name and the empty parentheses:

```
DrawLine()
```

This is a request for the code within the named function to be executed.

The Final Code

The complete program will now contain two sections. One section will contain the code for the function and the other section the main logic of the program.

The function code must be placed after the end of the main logic.

This gives us the code shown in FIG-8.1.

FIG-8.1

Using a Function

```

rem *** main program ***

DrawLine()
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine()
    Print("*****")
endfunction

```

Notice that the **end** statement has been added to emphasise the end of the main program logic.

Activity 8.2

Start a new project called *UsingFunctions*. Modify *main.agc* to match the code given in FIG-8.1. Test and save your project.

How the Code is Executed

When a call is made to a function, control transfers to that function, its code is executed, and then control returns to the line immediately following the original call to the function. FIG-8.2 shows the stages involved during the execution of the program shown above.

FIG-8.2

The Function Calling Mechanism

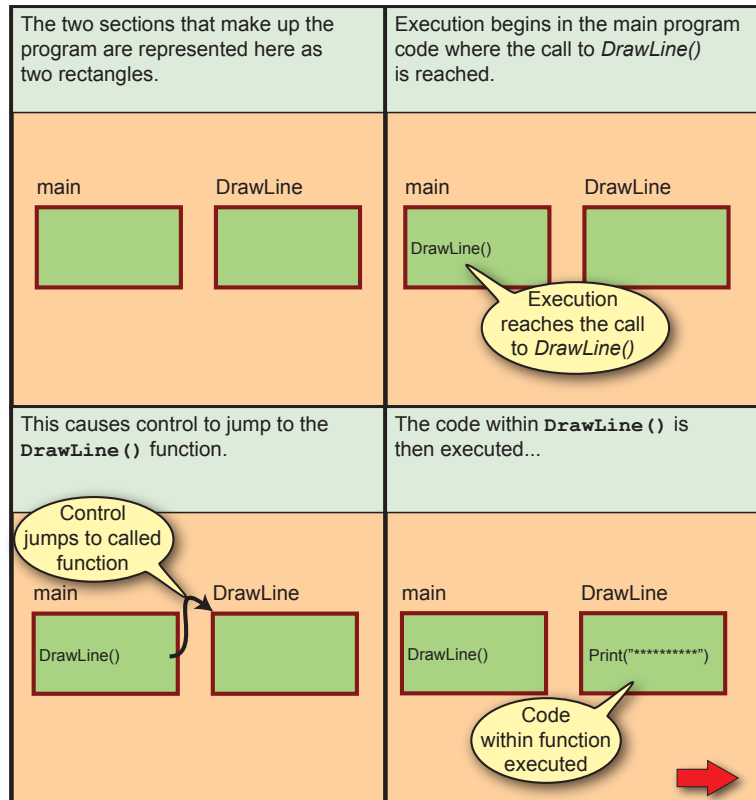
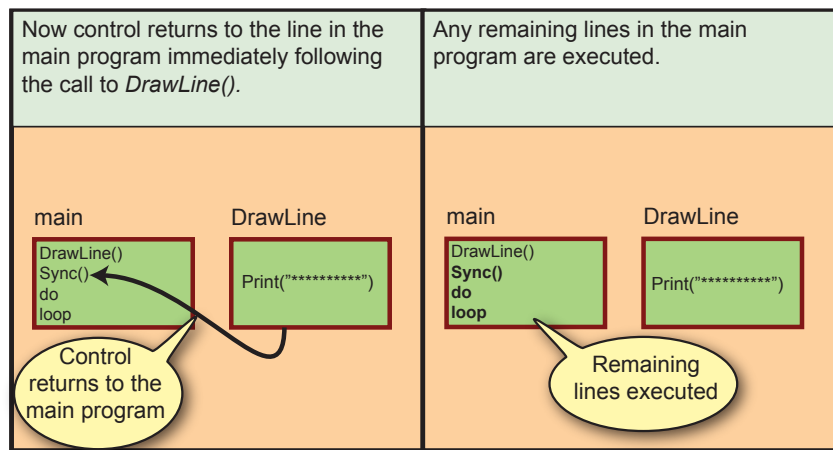


FIG-8.2

(continued)

The Function Calling Mechanism



A function can be called as often as required from any point in the main program logic.

So, if we wanted to draw two lines of asterisks, we would change the code for the main section to

```
DrawLine ()
DrawLine ()
Sync ()
do
loop
end
```

which adds a second call to the function.

Activity 8.3

Modify *UsingFunctions* so that two calls are made to *DrawLine()* as in the code shown above.

Test your program to check that two lines of asterisks are drawn. Resave your project.

Local Variables

We are at liberty to use variables within a function. The variables used within a function are local to that function and hence are known as **local variables**. When a variable is local to a function, it is allocated space within the computer's memory only while that function is being executed. Once execution finishes, the allocated memory space is freed up and the variable no longer exists.

Because a variable is local to a function, that means that we may use a variable within a function that has the same name as a variable in the main program logic without causing an error. The program will treat the two variables as separate entities. The program in FIG-8.3 demonstrates this use of a local variable sharing a name with a variable in the main part of the program.

The variable *v* in the main program is assigned the value 3 while *v* within the function is assigned the value 6. The **Print** statement within the function will display the value held in the local variable (6) while the **Print** statement in the main section will

FIG-8.3

Using Local Variables

print 3 - the value in the other variable named v.

```

rem *** Variable v in the main program ***
v = 3
Test()
Print(v)
Sync()
do
loop
end

function Test()
  rem *** Variable v local to the function ***
  v = 6
  Print(v)
endfunction

```

Activity 8.4

Create a new project called *LocalVariable* and change *main.agc* to contain the code shown in FIG-8.3.

Run the program and verify that each variable contains a different value.

Alternative Coding

As long as a function performs the task described within the mini-spec, then exactly how that result is achieved is up to the programmer. Back on the first page of Chapter 1 we saw that there is usually more than one algorithm for achieving a required result (the 4 litre problem). So let us look at another way of creating a line of 10 asterisks:

```

function DrawLine()
  for c = 1 to 10
    PrintC("*")
  next c
  Print(" ") //New line
endfunction

```

If we take a moment to look at the code above, we can see that the `for` loop will print the 10 asterisks - one at a time - and the final `Print()` statement will move the cursor onto a new line.

Activity 8.5

Modify *UsingFunctions*, replacing the existing code for *DrawLine()* with the new code given above.

Check that exactly the same results are produced as before. Resave your project.

Parameters

Sometimes a device needs to be supplied with information before it can perform its function. For example, you need to press a button on your TV remote to specify which channel you want to view. This same principle also holds for software functions: most functions need to be supplied with one or more values in order to determine exactly what is required of it. These values are known as **parameters**.

If we wanted to allow the length of the line created by *DrawLine()* to be specified when the function is called, we need to pass that information to the function in the form of a parameter.

To pass a parameter to a function, we need to rewrite the description of that function adding parameter details such as the parameter name and its type. In the case of the *DrawLine()* function, the new mini-spec would be:

FUNCTION NAME	:	DrawLine
PARAMETERS		
In	:	ilength : integer
DESCRIPTION	:	Draws a horizontal line of <i>ilength</i> asterisks

Notice that the parameter is described as an **in parameter**. This description is used because the value is being “given” to the function.

From our updated description we create the new code:

The parameter given in the function’s code is known as the **formal parameter**.

```
function DrawLine(ilength)
  for c = 1 to ilength
    PrintC("*")
  next c
  Print(" ")
endfunction
```

Notice that the parameter is placed within the parentheses of the first line of the function and that the parameter is then used as the end value in the **for** statement so that the loop now iterates *ilength* times.

And finally, the call made to the function from the main section of the program must supply a value for the parameter:

The parameter specified when the function is called is known as the **actual parameter**.

```
DrawLine(8)
```

This value will be copied into the function’s formal parameter, *ilength*, just before the code of the function is executed.

Activity 8.6

In *UsingFunctions*, modify *DrawLine()* to match the code given above.

Change the calls to *DrawLine()* so that a line of 5 asterisks followed by a line of 12 asterisks is displayed. Test and save your project.

The actual parameter passed to a function can be given as a constant (as in the example above), a variable or an expression. Hence, if we start by storing a number in a variable

```
num = Random(1,20)
```

we can pass the value held in that variable as the parameter to our function with the line

```
DrawLine(num)
```

or we can include a calculation within the function call

```
DrawLine(num * 2)
```

and the result of that calculation will be passed as the parameter value.

Activity 8.7

Modify *UsingFunctions* so that the main program generates a random number in the range 1 to 10, storing the result in a variable, *num*.

Change the parameters given in the calls to *DrawLine()* so that the first call uses *num* as the parameter and the second call uses *num*3 -2* as the parameter.

Test and save your project.

A final option is to use the value returned by one function as the parameter for another as in the line:

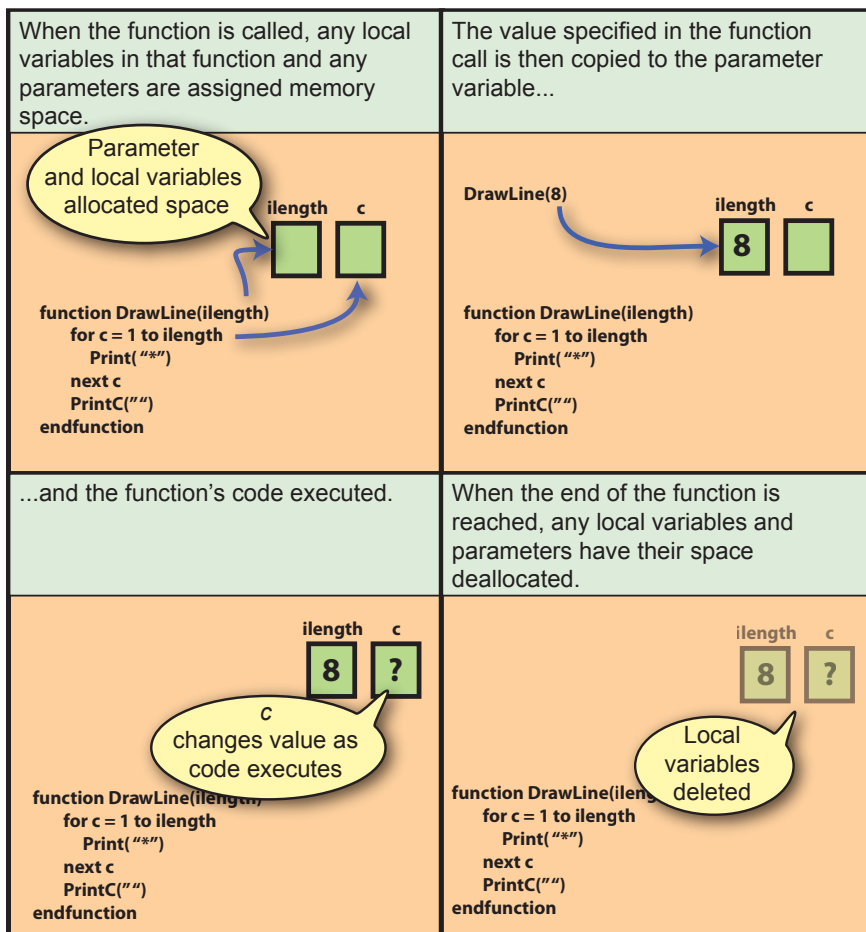
```
DrawLine(Random(1,10))
```

which will generate a random number in the range 1 to 10 and then use that value as the parameter to *DrawLine()*.

FIG-8.4 shows what's happening when a parameter is passed to a function.

FIG-8.4

How Parameter Passing Works



Multiple Parameters

A function can have as many parameters as required and these parameters can be of any type: integer, real or string.

To demonstrate this, we'll write yet another version of the *DrawLine()* function in which the character used to construct the line is also passed as a parameter. Of course, we start by updating the mini-spec:

FUNCTION NAME	:	DrawLine
PARAMETERS		
In	:	ilength : integer
	:	schar : string
DESCRIPTION	:	Draws a horizontal line of <i>ilength</i> characters. The character used in the construction of the line is <i>schar</i> .

From this we can create the modified function:

```
function DrawLine (ilength, schar$)
  for c = 1 to ilength
    PrintC(schar$)
  next c
  Print(" ")
endfunction
```

The \$ symbol is omitted from the parameter name since this is a BASIC requirement and not part of the design.

Notice that parameters are separated from each other by commas. The second parameter's name differs slightly from that in the mini-spec because of AGK BASIC's requirement that string variables must end with a dollar symbol (\$).

Now we need to supply two values when we call the function; one for the length of the line, the other for the character to be used in the construction of the line. A typical call would be:

```
DrawLine (12, "=")
```

which would assign the value 12 to the parameter *ilength* and the string "=" to *schar\$* and thereby produce a line on the screen created from twelve = symbols.

Activity 8.8

Modify your *DrawLine()* function so that the character used can be passed as a parameter.

Change the main section of the code so that a single line of 10 # characters is drawn.

It's important that you put the values in the correct order when you call up a function. For example, the line

```
DrawLine ("=" ,12)
```

would be invalid since the function expects the first value given in the parentheses to be an integer.

Pre-conditions

When a function uses a parameter, we will often need to restrict the range of values which may sensibly be assigned to that parameter. For example, when we specify what length of line we want *DrawLine()* to produce, it doesn't make sense to pass a negative value to the function. Equally, it makes little sense to request a line hundreds of characters long, since there is a limit to how many characters can fit on a single line of the app window. We might therefore expect the value specified for *ilength* to be in a range such as 1 to 100.

When we place limitations on the conditions under which a function can operate successfully, these limitations are known as the **pre-conditions** of the function.

We can therefore state that the pre-condition for *DrawLine()* is that the parameter *ilength* lies between 1 and 100.

We would start by adding this restriction to the mini-spec:

FUNCTION NAME	:	DrawLine
PARAMETERS		
In	:	ilength : integer schar : string
PRE-CONDITION	:	1 <= ilength <= 100
DESCRIPTION	:	Draws a horizontal line of <i>ilength</i> characters. The character used in the construction of the line is <i>schar</i> .

Now we need some way of enforcing this limit. We do this by adding an **if** statement at the beginning of the *DrawLine()* function which causes that function to be aborted if the value of *ilength* is outside the acceptable range.

exitfunction

The **exitfunction** statement is designed to be placed within a function. When executed, this statement causes the remaining statements in the function to be ignored, ending execution of the routine and returning to the code which called the function in the first place.

We can make use of this statement to terminate execution of a function when its parameter(s) fall outside an acceptable range. In the case of *DrawLine()*, we would add the following lines right at the start of the routine to check the value of *ilength*:

```
rem *** if ilength outside 1 to 100, terminate function ***
if ilength < 1 or ilength > 100
    exitfunction
endif
```

Activity 8.9

Modify your *DrawLine()* function so that if *ilength* is outside the range 1 to 100, the function terminates without anything being drawn.

Check that the new code works by attempting to draw a line 101 characters long constructed from the + symbol.

About Pre-Conditions

Be careful when choosing the pre-condition for a function. A pre-condition should only prevent situations which the function itself cannot handle. Do not create a pre-condition simply because you feel that using a value outside a range seems unreasonable to you. If the function's code can handle the situation then allow that situation to occur.

For example, we have stated in the pre-condition of *DrawLine()* that *ilength* must be in the range 1 to 100. But could the code handle values outside that range? The answer to this question is - yes.

Values of zero or less will simply mean that the `for` loop will iterate zero times and so the only output will be that caused by the `Print(" ")` statement which will move subsequent output onto the next line on the screen. Values greater than 100, the characters produced may spread over several lines, but the function will still work as described.

So we were probably mistaken to impose a pre-condition on *DrawLine()*.

When a function has no restrictions we would describe the pre-condition within the mini-spec as:

PRE-CONDITION : None

Return Types

Not only can we supply values to a function (in the form of parameters), but some functions also return results.

For example, let's assume we wish to create a function called *SumIntegers()* which takes an integer parameter, *ival*, and returns the sum of all the integer values between 1 and *ival*.

When describing such a routine in a mini-spec, we add the returning value as an **out parameter**.

FUNCTION NAME	:	SumIntegers
PARAMETERS		
In	:	ival : integer
Out	:	irestult : integer
PRE-CONDITION	:	None
DESCRIPTION	:	Sets <i>irestult</i> to the sum of the integers between 1 and <i>ival</i> .

To return a value from an AGK BASIC function we add the value to be returned immediately after the term `endfunction`. So *SumIntegers()* would be coded as:

```
function SumIntegers(ival)
    irestult = 0
    for c = 1 to ival
        irestult = irestult + c
    next c
endfunction irestult
```

Activity 8.10

Create a project called *TestFact* which contains a function named *Factorial* which implements the following mini-spec:

FUNCTION NAME	:	Factorial
PARAMETERS		
In	:	ival : integer
Out	:	irestult : integer
PRE-CONDITION	:	None
DESCRIPTION	:	Sets <i>result</i> to the product of the integers between 1 and <i>ival</i> . For example, if <i>ival</i> is 5 then <i>irestult</i> would be the value of 1 x 2 x 3 x 4 x 5.

The main program should generate a random number between 1 and 10, display that number, then display the result of *Factorial()* using the generated value as the *In* parameter.

Test and save your project.

When calling a function that returns a value, that value can be assigned to a variable, displayed, or used in an expression. Examples of valid calls to *SumIntegers()* are given below:

```
sum = SumIntegers(10)
Print(SumIntegers(5))
answer = SumIntegers(9)/3
if SumIntegers(no) < 100
```

Functions can also return a string value. For example, the function *FillString(ch\$, num)* returns a string containing *num* copies of *ch\$*:

```
function FillString(ch$,num)
    sresult$ = ""
    for c = 1 to num
        sresult$ = sresult$ + ch$
    next c
endfunction sresult$
```

and might be called with a line such as

```
h$ = FillString("H",10)
```

which would place a string containing 10 H's in the variable *h\$*.

Return Values and Pre-Conditions

Routines such as *SumIntegers()* and *Factorial()* cannot successfully return a result for all integer values, since the space assigned to a variable is of a fixed size so there would be insufficient space to hold the result produced by *SumIntegers(100)* or *Factorial(16)*. For this reason we need to impose a pre-condition in each case limiting the value of the *In* parameter.

Of course, this is easily done in the mini-spec. We could add the line

```
PRE-CONDITION   :   ival <= 50
```

in *SumIntegers*

and

```
PRE-CONDITION   :   1 <= ival <= 15
```

in *Factorial*.

The problem arises when we attempt to implement these restrictions in the code of the functions. We might be tempted to start *SumIntegers()* with the lines

```
if ival > 50
    exitfunction
```

but because *SumIntegers()* is designed to return a value, it is not legal to exit that function without returning a value.

This means that the `exitfunction` statement, as shown above, is not valid since it attempts to exit the function without returning a value. Luckily, the statement's format allows for a value to be specified after the keyword `exitfunction` and this value is returned by the function.

But that just leaves us with another problem - what value should we return when the routine does not meet its pre-conditions? We can return any value we like, but usually this is handled by returning a special value which cannot occur when the function's pre-conditions are met. For example, here we could return the value -1, since it is an impossible result to achieve when *ival* is less than 50. We would do this with the line:

```
exitfunction -1
```

This allows us to add back the pre-condition to our routine, the final version of the code being:

```
function SumIntegers(ival)
    rem *** Exit with -1 if pre-condition ***
    rem *** not met ***
    if ival > 50
        exitfunction -1
    endif

    ireresult = 0
    for c = 1 to ival
        ireresult = ireresult + c
    next c
endfunction ireresult
```

Activity 8.11

Modify your *Factorial()* function from the last Activity so that it implements the pre-condition that *ival* must lie between 1 and 15. The function should return zero if the parameter is outside this range.

Test the update by calling the function with the parameter value set to 16. Resave your project.

When a function such as *SumIntegers()* (which returns a dummy result if its precondition has not been met) is called, the main program must check that the function has performed correctly. This is done by making the main program check the value returned by the function. A typical piece of code for doing this is:

The parameter *number* is assumed to be a variable that has been assigned a value earlier in the program.

```
result = SumIntegers(number)
if result = -1
    Print("Could not calculate result")
else
    Print(result)
endif
```

Activity 8.12

In *TestFact*, change the main program to generate a number between 10 and 20. Attempt to find the factorial of the number generated, but if the number is over 15, display the message “Factorial too high to calculate.” along with the generated value.
Run your program so that at least one run produces the error message.

Returning a string from a function is no more difficult than returning a numeric value.

The program in FIG-8.5 contains a function which returns a random-length string of random letters.

FIG-8.5

Random Length String Function

```
rem *** Generate string ***
text$ = RandomString()
rem *** Display string ***
Print(text$)
Sync()
do
loop

rem *** Generate a random-length string of random letters ***
function RandomString()
    rem *** Generate length for string ***
    size = Random(1,50)
    rem *** start with empty string ***
    sresult$ = ""
    rem *** FOR size times ***
    for c = 1 to size
        rem *** Add new character to end of string ***
        sresult$ = sresult$ + Chr(Random(65,90))
    next c
    rem *** return the string generated ***
endfunction sresult$
```

This code makes use of an AGK function called *Chr()* which returns the character whose ASCII code matches the value of the parameter. More about this function in the next chapter.

Activity 8.13

Create a mini-spec for the function *RandomString()* given in FIG-8.5.

Create a new project called *StringFunction*. Edit *main.agc* to match the code given in FIG-8.5.

Test and save the project.

Function Flexibility

The more flexible a function, the more useful it is. For example, the final version of *DrawLine()* is much more flexible than the first, since it allows the length and construction character of the line to be specified when the function is called, whereas the first version could create only a line of exactly ten asterisks. With this added flexibility we could use the function to draw a simple bar chart for example - something not possible with the original version of the routine.

So, wherever possible, you should always try to add the maximum flexibility to any routine you create as long as this does not lead to over-complex code or unacceptable execution times.

We will add some flexibility to our *RandomString()* function with a new mini-spec:

FUNCTION NAME	:	RandomString
PARAMETERS		
In	:	ilength : integer
Out	:	sresult : string
PRE-CONDITION	:	ilength = -1 or 1 <= ilength <= 50
DESCRIPTION	:	IF ilength = -1 THEN sresult is a string of random capital letters of a random length between 1 and 50 ELSE sresult is a string of random capital letters exactly ilength characters long ENDIF

➡ Although a mini-spec may have a description written in structured English, this does not mean that the program must employ that logic to implement the routine.

Notice that the mini-spec's description makes use of structured English this time. A description can be written in any way you please; the only requirement is that it must be complete and unambiguous. A mini-spec is the document used by the programmer as a statement of exactly what a function must do, so that document must contain all the details required.

Activity 8.14

Modify the code for *RandomString()* in your *StringFunction* project so that it matches the mini-spec requirements given above. If *ilength* is not within the specified range, an empty string should be returned.

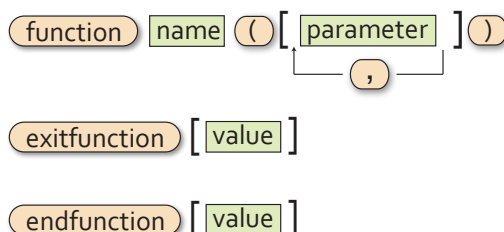
Test and save the project.

Statement Formats

We've introduced three new function-related statements in this section; the format of these are given in FIG-8.6.

FIG-8.6

function
exitfunction
endfunction



where:

name	is the name of the function. The name chosen must conform to the same rules used when creating variable names.
parameter	is the name of any value passed to the function. Names should be appropriate for the nature of the value being passed.
value	is the value returned by the function. This can be specified using a variable, constant or expression.

Summary

- A function is a named section of code.
- Functions should be relatively short and perform only a single task.
- Functions in AGK BASIC begin with the term **function** and end with the term **endfunction**.
- A function must be given a unique name.
- A function name should reflect the purpose of the function and must conform to the same rules as a variable name.
- A function can include zero or more *in* parameters.
- The parameter(s) listed in a function's code are known as the formal parameters.
- A function can return a single value.
- Any variables used within a function are local to that function.
- Local variables may have the same name as variables in the main program without causing an error.
- Before being coded, the details of a function should be documented in a mini-spec.
- Where a parameter's value must fall within a given range, this should be stated as a pre-condition in a function's mini-spec.
- Any pre-condition is tested by an **if** statement at the start of the function.
- When a pre-condition is not fulfilled, a function should exit without executing the main part of its code.
- Exiting a function before all of its code has been executed is achieved using the **exitfunction** statement.
- Where a pre-condition is not met and the function is designed to return a value, some error-indicating value should be returned.
- A function is called by giving the function name, parentheses and, where required, a list of parameter values.
- The parameters given when calling a function are known as the actual parameters.
- The value returned by a function can be assigned to a variable, displayed, used in an expression or used as the parameter to another function call.

BASIC Subroutines

Introduction

Using functions is the best way to create modular software in AGK BASIC, but the language does offer another way to achieve a similar effect, and that is to use subroutines. Although we've used the term subroutine earlier to mean any modular section of code, in AGK BASIC the word has a more specific meaning as we'll see below.

Creating a Subroutine

The original version of the BASIC language (invented in 1964) had no provision for true functions as described earlier in this chapter. Instead it made use of two statements, `gosub` and `return` which allowed a section of code to be executed and then a return made to the point of call. In this respect it was similar to a true function, but there was no way to pass parameter values or make use of local variables.

Although of limited usefulness, the `gosub` and `return` statements have been retained in AGK BASIC and so a description of how these statements are used is included here.

In order to compare true functions with the subroutine approach of `gosub`, we will recode the *DrawLine* routine using this older approach.

The start of a subroutine is marked with a label giving the name of the subroutine. This will be the name given in the mini-spec.

A label is just a valid name followed by a colon, for example:

```
DrawLine:
```

This is followed by the code

```
DrawLine:
  for c = 1 to 10
    Print("**")
  next c
  Print(" ")
```

and finally, the `return` statement:

```
DrawLine:
  for c = 1 to 10
    Print("**")
  next c
  Print(" ")
  return
```

To execute the code, we use the `gosub` statement giving the name of the label we used to start the code:

```
gosub DrawLine
```

A complete program implementing this example is shown in FIG-8.7.

FIG-8.7

Using Subroutines

```
rem *** Using GOSUB ***
gosub DrawLine
Sync()
do
loop
end

DrawLine:
  for c = 1 to 10
    PrintC("*")
  next c
  Print(" ")
  return
```

Activity 8.15

Start a new project called *TestGosub*.

Edit *main.agc* to match the code in FIG-8.7.

Test and save your project.

It is perhaps worth pointing out that no modern language offers this simplistic method of implementing modular programming because of the restrictions it imposes. Although you may see some examples of the `gosub` statement in action, these will be in very simple programs. So...

avoid using `gosub` - stick to proper functions!

A Library of Functions

Introduction

Most functions will be designed for a specific project and will only ever be used in that project, but a more general-purpose routine can be re-used in different programs.

We have already experienced this when we used the three Button functions back in earlier chapters to allow us to enter integer values.

Creating a Library

If we identify one or more routines that might be useful later, then we need to copy these routines into a new *agc* file which contains nothing but the code for these selected functions.

If you are building a library of reusable routines, the best approach is to create a separate *agc* file for each category. For example, we might keep all the math functions in one file and all the string-handling functions in another. Other functions which fall into an existing category can be added to the appropriate file later.

FIG-8.8 shows you how to place the *RandomString()* function in a separate file.

FIG-8.8

Creating a Library

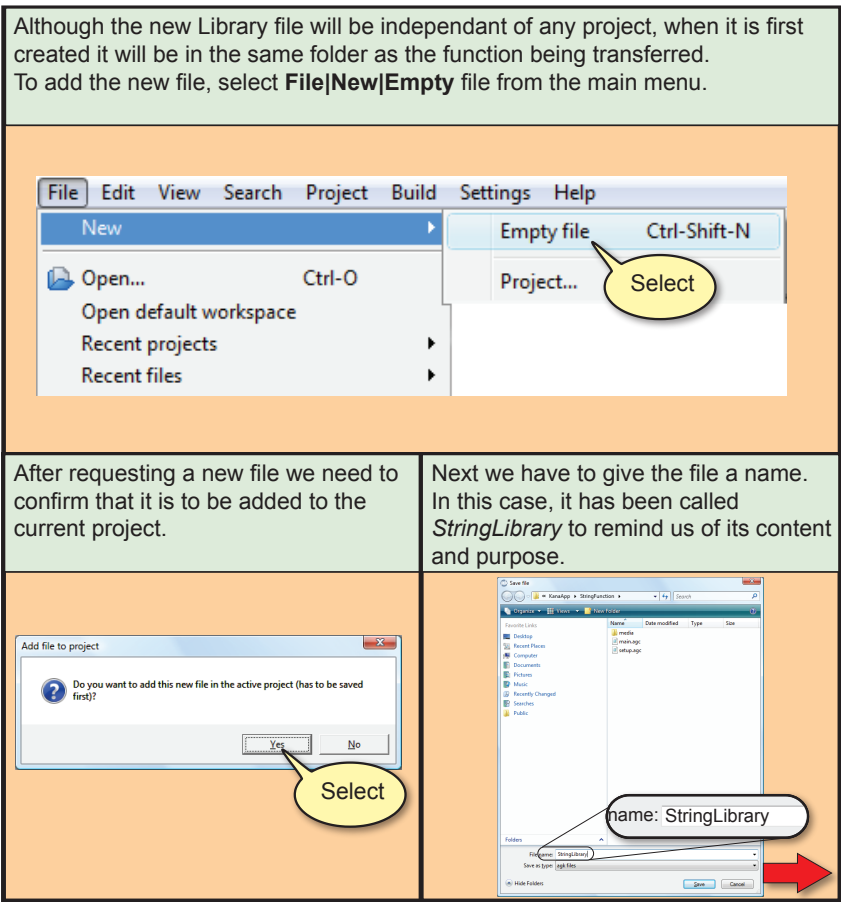
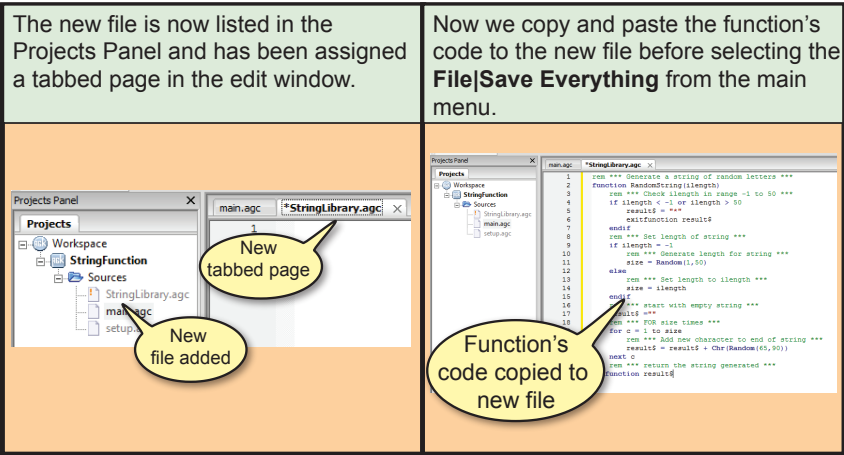


FIG-8.8

(continued)

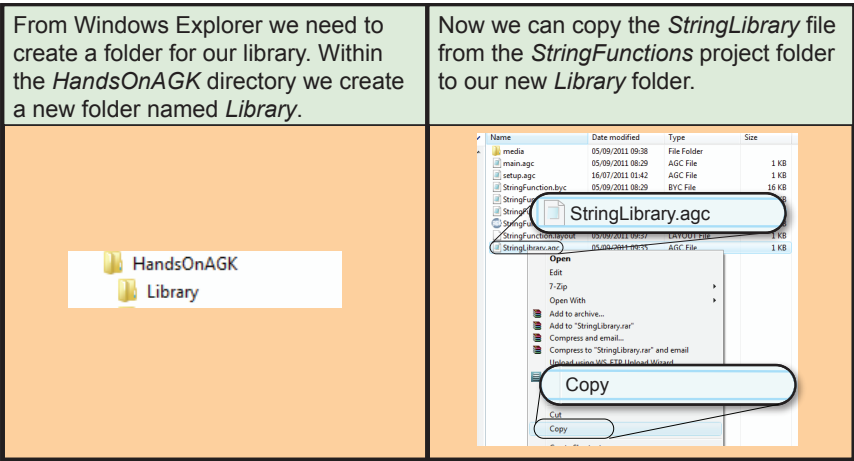
Creating a Library



Since the point of creating this library file is to make use of the functions it contains in other projects, it makes sense to remove the file from the current project's folder and store it in a more generalised one (see FIG-8.9).

FIG-8.9

Moving the Library File



Activity 8.16

Open your *StringFunction* project.

Create a new file called *StringLibrary*.

Copy the code for the function *RandomString()* from *main.agc* into the new file.

Save all the files within the project then close the project.

Open Windows Explorer and create a new subfolder called *Library* within the *HandsOnAGK* folder.

Copy the file *StringLibrary* from the *StringFunction* folder to the *Library* folder.

Creating Modular Software

Introduction

Now that we know the basic techniques required to design and implement functions in AGK BASIC, we're ready to rewrite the *SpotTheDifference* project using functions. We'll start by repeating the structured English description of the game:

- 1 Load resources
- 2 Set up game screen
- 3 Play game
- 4 End game

This is a good guide for identifying the routines needed within the program. The mini-spec for each identified routine is shown below.

FUNCTION NAME	: LoadResources
PARAMETERS	
In	: None
PRE-CONDITION	: None
DESCRIPTION	: The images <i>Button.bmp</i> , <i>Credits.jpg</i> , <i>End.jpg</i> <i>Main.jpg</i> , <i>Ring.png</i> , <i>Fail.jpg</i> sound file <i>Click.wav</i> and music file <i>Backgroundmusic.wav</i> are loaded and assigned ID numbers.

FUNCTION NAME	: SetUpGameScreen
PARAMETERS	
In	: None
PRE-CONDITION	: None
DESCRIPTION	: Start music playing. The image <i>main.jpg</i> is displayed. The rings are positioned at each difference in the right-hand picture and then hidden.

FUNCTION NAME	: PlayGame
PARAMETERS	
In	: None
Out	: timetaken : integer
PRE-CONDITION	: None
DESCRIPTION	: The user selects areas within the right-hand picture. If the area is within a hidden ring, the ring is displayed. The game ends when all six rings are displayed or when 7 areas have been selected. All resources used by this routine are deleted when play is complete. The routine returns the number of seconds taken to complete the game.

FUNCTION NAME	: EndGame
PARAMETERS	
In	: timetaken : integer
PRE-CONDITION	: None
DESCRIPTION	: Sets the aspect ratio to portrait mode. If the player has selected all six differences, then the <i>End</i> screen is displayed along with the time taken to find all the differences. If all differences were not found, the <i>Fail</i> screen is displayed. Both screens have a button which when pressed displays the <i>Credits</i> screen for 5 seconds before returning to the previous screen.

Now we're ready to start turning our design into a program.

There are various ways to tackle this. If we had several people working on the program, we could give each a separate routine to work on at the same time. It would then just be a matter of bringing together the separate routines to give us the final program. On the other hand, if only one person is working on the coding, the routines are coded one after another, usually starting with the main logic.

Top-Down Programming

When we code our routines one at a time, starting with the main logic, this is known as **top-down programming**. The name is used because we start with the main part of the program (the top) and then work our way through the routines called by that main part. We'll see how it's done below.

Step 1

We start by turning the outline logic given in the structured English into AGK BASIC code. An important point to note is that the program code must match that logic exactly. If we find we have to deviate from this logic, then we must go back and modify the details given in the structured English.

Actually, the code for the main section of the program becomes little more than a set of calls to the other routines:

```
LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
```

Notice that the main code really doesn't do any of the detailed work, it leaves that to the routines. The main section only has to call up each of the routines in the correct order and save any values returned by one function to pass it on to another function.

Activity 8.17

Start a new project called *SpotTheDifference2*.

Compile the default code and copy the required files into the *media* folder. Modify *main.agc* so that it contains the four lines given above.

Edit *startup.agc* setting width to 1024 and height to 768.

Step 2

To get the main logic to run, we must write code for the routines that are called. And yet, if we do that, it would appear that the whole program will need to be completed before the program can be executed for the first time.

The way round this problem is to write almost empty routines with the required names as shown below:

```
function LoadResources
    Print("LoadResources")
endfunction

function SetUpGameScreen()
    Print("SetUpGameScreen")
endfunction

function PlayGame()
    Print("PlayGame")
endfunction 10

function EndGame(timetaken)
    Print("EndGame")
endfunction
```

Take a moment to look at this code. Each function displays its own name, takes any necessary parameters and returns a value where necessary. We need to make sure that the names, parameter names and return types match with those given in the mini-specs.

These empty routines are known as **test stubs** and are written so that we can test the main logic without having the final code for any of the routines which that logic has to call.

Activity 8.18

In *SpotTheDifference2*, add an **end** statement after the existing four lines of code. This will separate the main logic from the code for the functions.

Add the four test stubs given above to your program.

Add a **Sync()** statement and **do...loop** structure immediately before **end** so that the messages displayed by the function will appear on the screen.

Run and save your program.

By running the program, we can see that the functions are executed in the correct order.

Step 3

Now we can begin to remove the stubs in our project and replace them with the final version of each routine. As each new routine is added the program is tested to make sure that the new routine, and the program as a whole, are working correctly.

The code for the first routine, *LoadResources()*, is given below:

```

rem *** Load resources ***
function LoadResources()
    rem *** Load images ***
    main =      LoadImage("Main.jpg")
    finish =    LoadImage("End.jpg")
    credits =   LoadImage("Credits.jpg")
    ring =      LoadImage("Ring.png",0)
    button =    LoadImage("Button.bmp",1)
    fail =      LoadImage("Fail.jpg")
    rem *** Load sounds ***
    ringsound = LoadSound("Click.wav")
    rem *** Load music ***
    backgroundmusic = LoadMusic("Background.wav")
endfunction

```

Activity 8.19

Remove the *LoadResources()* test stub from your program and substitute the complete function as shown above.

Run the program again. Although no new output is produced, there will be an error message if any of the files cannot be found. Save your project.

The second routine, *SetUpGameScreen()*, is coded as:

```

rem *** Set up main section of the game ***
function SetUpGameScreen()
    rem *** Play music ***
    PlayMusic(backgroundmusic)
    rem *** Show main screen ***
    CreateSprite(1,main)
    SetSpriteSize(1,100,100)
    rem *** Load rings at image differences ***
    CreateSprite(2,ring)
    SetSpriteSize(2,-1,10)
    SetSpritePosition(2,91,86)
    CloneSprite(3,2)
    SetSpritePosition(3,51.5,22)
    CloneSprite(4,2)
    SetSpritePosition(4,49,68)
    CloneSprite(5,2)
    SetSpritePosition(5,73,66)
    CloneSprite(6,2)
    SetSpritePosition(6,88.5,66)
    CloneSprite(7,2)
    SetSpritePosition(7,55.75,62.5)
    rem *** Hide rings ***
    for c = 2 to 7
        SetSpriteDepth(c,9)
        SetSpriteVisible(c,0)
    next c
    rem *** Update screen ***
    Sync()
endfunction

```

Activity 8.20

Add the complete version of *SetUpGameScreen()* to your project.

Run the program again. What problem occurs? Save your project.

Global Variables

The problem with *SetUpGameScreen()* is that it needs to make use of the IDs which were assigned to various resources by the *LoadResources()* function.

The *LoadResources()* function assigned the resource IDs to variables such as *backgroundmusic*, *main* and *ring*. But these variables are local to that routine, so when we mention variables of the same name in *SetUpGameScreen()* the program doesn't realise that we are trying to refer to the same variables as those in the earlier routine. Instead, we get a new set of variables that are local to *SetUpGameScreen()* and these new variables do not contain the values we need.

If only one value had been needed to be shared between the routines, we might have made use of a return value from the first routine and a parameter to the second (note that this is exactly how the *timetaken* is passed between *PlayGame()* and *EndGame()*).

However, since so many ID values need to be shared between *LoadResources()* and the other routines, we have no choice but to store these values in **global** variables.

Global variables are exactly the opposite from local variables. Whereas local variables exist only within the routine in which they are used, global variables exist throughout the program and can be referred to anywhere within the program.

To declare a global variable, we need to start with the keyword **global** and then give the variable name. In this program, we want the variables that contain the IDs of the various resources to be global - that way we can refer to them in any of the functions. So the code needed is:

```
rem *** Define global variables ***
rem *** IDs for images ***
global main, finish, credits, ring, button, fail
rem *** IDs for sound ***
global ringsound
rem *** IDs for music ***
global backgroundmusic
```

This code is placed right at the start of the main logic, before the function calls.

Now the term *main* in *LoadResources()* and *main* in *SetUpGameScreen()* refer to the same global variable.

Activity 8.21

Add the global declarations to your code and run the program again.

Does the program run correctly this time? Save your project.

The third function, *PlayGame()* is coded as:

```
rem *** Play game ***
function PlayGame()
    rem *** Start timer ***
    start = GetSeconds()
    CreateText(1, str(timetaken))
    SetTextColor(1, 0, 0, 255)
    SetTextPosition(1, 88, 6)
    rem *** Set count of differences found ***
```

```

found = 0
rem *** Number of clicks so far is zero ***
rem *** Get user clicks until all 6 differences found ***
repeat
    rem *** Check for clicked(pressed)
    pressed = GetPointerPressed()
    rem *** IF pressed, ***
    if pressed = 1
        rem *** Add 1 to clicks ***
        inc presscount
        rem *** Check for sprite hit ***
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF clicked over hidden ring THEN
        if hit > 1 and hit <=7 and GetSpriteVisible(hit)=0
            rem *** Show ring ***
            SetSpriteVisible(hit,1)
            rem *** Play sound effect ***
            PlaySound(ringsound)
            rem *** Add 1 to differences found ***
            found = found + 1
        endif
    endif
    rem *** Update time so far ***
    timetaken = GetSeconds() - start
    SetTextString(1,Str(timetaken))
    Sync()
until found = 6 or presscount = 7
rem *** Delete existing sprites ***
for c = 1 to 7
    DeleteSprite(c)
next c
rem *** Delete sound ***
DeleteSound(ringsound)
rem *** Delete text ***
DeleteText(1)
endfunction timetaken

```

Activity 8.22

Add the third function to your project and check that it operates correctly.

Save your project.

The final function, *EndGame()* is coded as:

```

rem *** Finish game ***
function EndGame(timetaken)
    rem *** Wait before finishing ***
    Sleep(1000)
    rem *** Reset aspect ratio ***
    SetDisplayAspect(768.0/1024.0)
    rem *** IF all differences found ***
    if found = 6
        rem *** Show End screen... ***
        CreateSprite(1,finish)
        SetSpriteSize(1,100,100)
        rem *** ...and total time taken ***
        CreateText(1,str(timetaken))
        SetTextColor(1,0,0,0,255)
        SetTextPosition(1,36,31)
    endif
endfunction

```

```

        Sync()
    else
        rem *** Show Fail screen... ***
        CreateSprite(1,fail)
        SetSpriteSize(1,100,100)
    endif
    rem *** ... with button... ***
    CreateSprite(2,button)
    SetSpriteSize(2,15,-1)
    SetSpritePosition(2,80,90)
    rem *** Allow for Credits button press ***
    do
        pressed = GetPointerPressed()
        if pressed = 1
            x = GetPointerX()
            y = GetPointerY()
            hit = GetSpriteHit(x,y)
            rem *** IF Credit button pressed THEN ***
            if hit =2
                rem *** Show credits for 5 secs ***
                CreateSprite(3,credits)
                SetSpriteSize(3,100,100)
                SetSpriteDepth(3,8)
                Sync()
                Sleep(5000)
                rem *** Remove Credits screen ***
                DeleteSprite(3)
            endif
        endif
    Sync()
    loop
endfunction

```

Activity 8.23

Add the final function to your project and remove the `Sync()`, `do` and `loop` lines from the main section.

Test the completed project. Does it operate correctly?

The problem is that a variable whose value is determined in `PlayGame()` is required in `EndGame()`. What variable is this? To solve this problem, make the variable required in `EndGame()` a global variable.

Retest your project. Does it operate correctly? Save your project.

Activity 8.24

Now that the app is working correctly on your PC, it's time to try running it on another platform.

Make sure the app player or viewer is running on your device.

Press AGK's **Compile and Broadcast** button. The app should now start playing on your device. Does the app run correctly on the device?

The final problem we have to fix is to set the correct aspect ratio for the main game screen. Although we set the dimensions of the app window in *setup.agc*, when the app is running on your device, that setting has no relevance, so we need to add another `SetDisplayAspect()` statement to the *SetUpGameScreen()* function.

Activity 8.25

Modify the *SetUpGameScreen()* function so that it starts with the lines

```
rem *** Set screen aspect ***  
SetDisplayAspect(1024.0/768)
```

Save the updated project.

Press the **Compile and Broadcast** button and check that the project now runs correctly.

Global Variables and Mini-Specs

As a general rule, we should try to avoid the use of global variables. Global variables make functions less independent of each other since those functions share access to the global variables. Global variables can also make finding bugs in a program more difficult since almost any of the routines could be assigning invalid values to those variables.

However, there are times when global variables will be necessary (as is the case with the resource IDs if we load all the resources in a single function).

When a program does contain global variables, then those global variables should be listed and described as part of the documentation along with the mini-specs.

GLOBAL VARIABLES in SpotTheDifference

Image IDs	
main, finish, credits, ring, button, fail	: INTEGER
Sound IDs	
ringsound	: INTEGER
Music IDs	
backgroundmusic	: INTEGER
Number of differences found in image	
found	: INTEGER

Notice that the descriptions given give the purpose, name and type of any global variables.

Any routine that accesses global variables should include details of this. When a routine makes use of the current contents of a global variable, but does not change those contents, then the routine is said to **read** the variable. If a routine changes the contents of a global variable then this is known as a **write**.

Details of global variables read or written are added to a mini-spec after the parameter

details. So our updated mini-spec for *LoadResources* is:

FUNCTION NAME	:	LoadResources
PARAMETERS	:	
In	:	None
GLOBALS	:	
Read	:	None
Written	:	button, credit, finish, main, ring, fail, ringsound, backgroundmusic
PRE-CONDITION	:	None
DESCRIPTION	:	The images <i>Button.bmp, Credits.jpg, End.jpg</i> <i>Main.jpg, Ring.png, Fail.jpg</i> sound file <i>Click.wav</i> and music file <i>Backgroundmusic.wav</i> are loaded and assigned ID numbers.

Activity 8.26

Update the other mini-specs for the *SpotTheDifference2* project to give details of any global variables referenced in each of the routines.

Bottom-Up Programming

Top-down programming is particularly suited to a single programmer working alone, but if you're working as part of a team of programmers, then you're likely to get landed with having to code a specific routine which, when completed, will be handed over to the team leader. He will then add your routine to the main program.

So let's assume we've just been landed with the job of writing the *EndScreen* function. How do we go about doing this task without having the other parts of the program?

Well, we need to start by getting hold of the mini-spec for the routine and turning it into a coded function.

Although we might be tempted to think our job is done when we have coded the function, we really need to check that our routine is operating correctly. It won't do your reputation as a programmer any good if you hand over code which contains obvious faults.

To test a function we start by making sure that what we've written conforms to the requirements of the mini-spec. Once we're happy with that, then the code itself must be tested. Since a function only executes when called by another piece of code, we need to write a main program which will call up the function we want to test. This main program, known as a **test driver**, needs to perform five main tasks:

- Set up any resources required by the function
- Supply a value for any parameters required by the function
- Execute the function
- Display the value of any parameters passed to the function

➤ Display any value returned by the function

Sometimes testing a function on its own is going to be difficult since it is so dependent on the existence of other functions. In the case of *EndScreen()* we need to make sure the appropriate images have been loaded and assign a value to the global variable *found*. The test driver for *EndScreen()* is shown in FIG-8.10.

FIG-8.10

EndGame() Test Driver

```
rem *** EndGame Test Driver ***

rem *** Set up required resources ***
rem *** Global variables required ***
global finish, credits, button, fail
global found
rem *** Images required ***
finish = LoadImage("End.jpg")
credits = LoadImage("Credits.jpg")
button = LoadImage("Button.bmp",1)
fail = LoadImage("Fail.jpg")

rem *** Set found ***
found = 6
rem *** Call function under test ***
EndGame(10)
end

rem *** Finish game ***
function EndGame(timetaken)
    rem *** Set screen aspect ***
    SetDisplayAspect(768/1024.0)
    rem *** IF all differences found ***
    if found = 6
        rem *** Show End screen... ***
        CreateSprite(1,finish)
        SetSpriteSize(1,100,100)
        rem *** ...and total time taken ***
        CreateText(1,str(timetaken))
        SetTextColor(1,0,0,0,255)
        SetTextPosition(1,36,31)
        Sync()
    else
        rem *** Show Fail screen... ***
        CreateSprite(1,fail)
        SetSpriteSize(1,100,100)
    endif
    rem *** ... with button... ***
    CreateSprite(2,button)
    SetSpriteSize(2,15,-1)
    SetSpritePosition(2,80,90)
    rem *** Allow for Credits button press ***
    do
        pressed = GetPointerPressed()
        if pressed = 1
            x = GetPointerX()
            y = GetPointerY()
            hit = GetSpriteHit(x,y)
            rem *** IF Credit button pressed THEN ***
            if hit =2
                rem *** Show credits for 5 secs ***
                CreateSprite(3,credits)
                SetSpriteSize(3,100,100)
                SetSpriteDepth(3,8)
```



FIG-8.10
(continued)
EndGame() Test Driver

```

        Sync ()
        Sleep (5000)
        rem *** Remove Credits screen ***
        DeleteSprite (3)
    endif
endif
Sync ()
loop
endfunction
```

The coding shown here will test the routine working on the assumption that all 6 differences were found and that the total time taken was 10 seconds. To create a complete set of tests, we need to try various other times to ensure they are displayed correctly and also to set *found* to a value less than 6 which will check that the *Fail* screen is displayed correctly. Also, when the *End* and *Fail* screens are showing, we should press the **Credits** button to check that the *Credits* screen appears correctly.

Activity 8.27

- Start a new project called *EndScreenTestDriver*.
- Create the project’s *media* folder and copy the relevant resources to the folder.
- Change *main.agc* to match the code in FIG-8.10 and run the code.
- Make changes to the code so that other times are used and that the *Fail* screen is displayed rather than the *End* screen.

While various programmers worked on creating the various routines of a project, the project leader would create the code for the main program, making use of a set of test stubs for the functions called. As each function became available from the rest of the team he would replace each test stub with the actual function code and test the program as each new routine was added.

This approach to program construction is known as **bottom-up programming**.

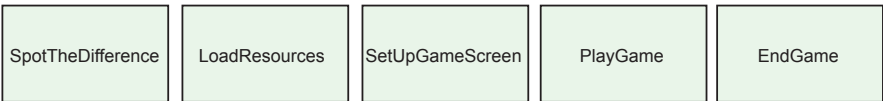
Structure Diagrams

As we begin to develop more complex programs containing several routines, it can be useful to retain an overview of the program’s structure showing which routine is called by which, and the values that pass between them. This is done using a structure diagram.

A structure diagram contains one rectangle for each routine in a program, including a rectangle representing the main program code (this is given the name of the project). Each rectangle contains the name of the routine it represents. The collection of rectangles for the *SpotTheDifference2* project is shown in FIG-8.11.

FIG-8.11
Function Rectangles

In the design, the program is referred to as **SpotTheDifference** (the 2 being removed).



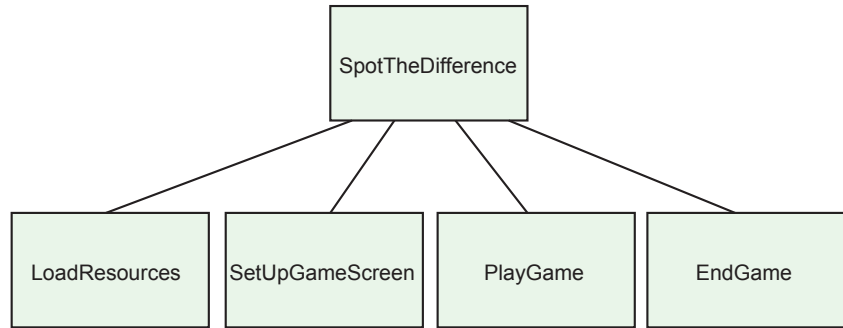
The rectangles are now set in a series of levels, with *SpotTheDifference* (the renamed

main logic) at the top. On the second level are routines called by *SpotTheDifference*.

The new layout is shown in FIG-8.12.

FIG-8.12

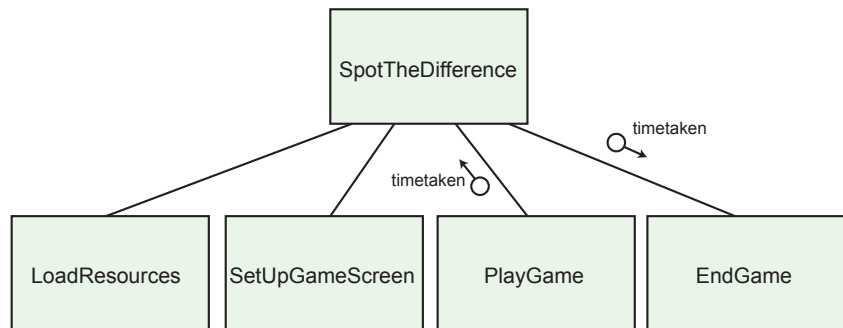
A Structure Diagram
Showing Call Structure



Finally, we add any parameters passed between the routines. In this case *PlayGame* returns the time taken to find all the differences and *EndGame* takes that same value as an *In* parameter. Parameters are represented by labelled, directed circles (see FIG-8.13).

FIG-8.13

A Structure Diagram
Showing Parameters



Note that global variables are not represented in structure diagrams.

The circle with arrowed line symbol in the diagram is used to show the direction in which data is passed, with *In* parameters pointing towards a routine and *Out* parameters pointing away from the routine.

Summary

- Good programming technique requires program code to be partitioned into routines.
- Each routine should perform a single task.
- A routine's name should reflect the purpose of that routine.
- Mini-specs should be produced when designing a routine.
- A mini-spec should include the name of the routine, its parameters, restrictions of the range of values a parameter may take and a detailed description of the routine's purpose.
- A routine should be made as flexible as possible so that it can be used in situations which differ slightly from the original requirement.
- The term **global** can be used to create a variable which can be accessed anywhere within a program.
- Top-down programming begins by coding the main routine.
- Top-down programming uses stubs in place of completed routines.

- Bottom-up programming starts by coding individual routines.
- Bottom-up programming uses test drivers to check that completed routines are operating correctly.
- A structure diagram shows every routine in a program, how they are called, and the values that pass between them.

Solutions

Activity 8.1

We could include the size, font and colour to be used for the asterisks.

Activity 8.2

No solution required.

Activity 8.3

Code for modified version of *UsingFunctions*:

```
rem *** main program ***
DrawLine()
DrawLine()
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine()
  Print("*****")
endfunction
```

Activity 8.4

The program's output is

6
3

The first of these is the value of the variable *v* defined within the *Test()* function; the second is the value held in the main program's *v*.

Activity 8.5

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***
DrawLine()
DrawLine()
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine()
  for c = 1 to 10
    PrintC("*")
  next c
  Print(" ")
endfunction
```

Activity 8.6

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***
DrawLine(5)
DrawLine(12)
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine(ilenlength)
  for c = 1 to ilenlength
    PrintC("*")
  next c
  Print(" ")
endfunction
```

Activity 8.7

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***

rem *** Generate random number ***
num = Random(1,10)
rem *** Call function ***
DrawLine(num)
DrawLine(num*3-2)
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine(ilenlength)
  for c = 1 to ilenlength
    PrintC("*")
  next c
  Print(" ")
endfunction
```

Activity 8.8

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***

rem *** Call function ***
DrawLine(10,"#")
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine(ilenlength, schar$)
  for c = 1 to ilenlength
    PrintC(schar$)
  next c
  Print(" ")
endfunction
```

Activity 8.9

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***

rem *** Call function ***
DrawLine(101,"+")
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine(ilenlength, schar$)
  if ilenlength < 1 or ilenlength > 100
    exitfunction
  endif
  for c = 1 to ilenlength
    PrintC(schar$)
  next c
  Print(" ")
endfunction
```

Activity 8.10

Code for *TestFact*:

```
rem *** main program ***
rem *** Generate random number ***
num = Random(1,10)
rem *** Find factorial of number generated ***
answer = Factorial(num)
rem *** Display results ***
PrintC("Factorial of ")
PrintC(num)
PrintC(" is ")
PrintC(answer)
Sync()
do
loop
end
```

```
rem *** Factorial Function ***
function Factorial(ival)
    irect = 1
    for c = 2 to ival
        irect = irect * c
    next c
endfunction irect
```

Activity 8.11

Code for the updated version of *TestFact*:

```
rem *** main program ***

answer = Factorial(16)
rem *** Display results ***
PrintC("Factorial of 16 is ")
Print(answer)
Sync()
do
loop
end

rem *** Factorial Function ***
function Factorial(ival)
    if ival < 1 or ival > 15
        exitfunction 0
    endif
    irect = 1
    for c = 2 to ival
        irect = irect * c
    next c
endfunction irect
```

Activity 8.12

Code for the updated version of *TestFact*:

```
rem *** main program ***
rem *** Generate random number ***
num = Random(10,20)
answer = Factorial(num)
if answer = 0
    rem *** Display error message ***
    PrintC(num)
    Print(" Factorial too high to calculate")
else
    rem *** Display results ***
    PrintC("Factorial of ")
    PrintC(num)
    PrintC(" is ")
    Print(answer)
endif
Sync()
do
loop
end

rem *** Factorial Function ***
function Factorial(ival)
    if ival < 1 or ival > 15
        exitfunction 0
    endif
    irect = 1
    for c = 2 to ival
        irect = irect * c
    next c
endfunction irect
```

Activity 8.13

FUNCTION NAME	:	RandomString
PARAMETERS		
In	:	None
Out	:	sresult : string
PRE-CONDITION	:	None
DESCRIPTION	:	Creates a string of random capital letters between 1 and 50 characters in length.

Activity 8.14

Code for the updated version of *StringFunction*:

```
rem *** Main program ***
rem *** Generate string ***
text1$ = RandomString(-1)
text2$ = RandomString(10)
text3$ = RandomString(-5)
rem *** Display strings ***
Print("Random length:" + text1$)
Print("Length 10 : " + text2$)
Print("Invalid : " + text3$+"XXX")
Sync()
do
loop

rem *** Generate a random-length string of random
letters ***
function RandomString(ilenlength)
    rem *** IF invalid length, return empty string
    ***
    if ilenlength <>-1 and (ilenlength <1 or ilenlength >50)
        exitfunction ""
    endif
    rem *** Determine length of string ***
    if ilenlength = -1
        rem *** Generate length for string ***
        size = Random(1,50)
    else
        size = ilenlength
    endif
    rem *** start with empty string ***
    sresult$ = ""
    rem *** FOR size times ***
    for c = 1 to size
        rem *** Add new character to end of string
        ***
        sresult$ = sresult$ + Chr(Random(65,90))
    next c
    rem *** return the string generated ***
endfunction sresult$
```

Notice that the third `Print` statement in the main section adds XXX to the display. This is used to prove that the returned string is empty rather than filled with space characters. For an empty string, the final colon and XXX will be joined (:XXX); if the string contained spaces there would be a gap between the colon and the X's (: XXX).

Activity 8.15

No solution required.

Activity 8.16

No solution required.

Activity 8.17

Code for *SpotTheDifference2*:

```
LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
```

Changes to *setup.agc*:

```
width=1024
height=768
```

Activity 8.18

Code for the updated version of *SpotTheDifference2*:

```
LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end
```

```

function LoadResources
    Print("LoadResources")
endfunction

function SetUpGameScreen()
    Print("SetUpGameScreen")
endfunction

function PlayGame()
    Print("PlayGame")
endfunction 10

function EndGame(timetaken)
    Print("EndGame")
endfunction

```

You should see the names of the four functions appear when the program is run.

Activity 8.19

Code for the updated version of *SpotTheDifference2*:

```

LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end

rem *** Load resources ***
function LoadResources()
    rem *** Load images ***
    main = LoadImage("Main.jpg")
    finish = LoadImage("End.jpg")
    credits = LoadImage("Credits.jpg")
    ring = LoadImage("Ring.png",0)
    button = LoadImage("Button.bmp",1)
    fail = LoadImage("Fail.jpg")
    rem *** Load sounds ***
    ringsound = LoadSound("Click.wav")
    rem *** Load music ***
    backgroundmusic = LoadMusic("Background.wav")
endfunction

function SetUpGameScreen()
    Print("SetUpGameScreen")
endfunction

function PlayGame()
    Print("PlayGame")
endfunction 10

function EndGame(timetaken)
    Print("EndGame")
endfunction

```

Activity 8.20

Code for the updated version of *SpotTheDifference2*:

```

LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end

rem *** Load resources ***
function LoadResources()
    rem *** Load images ***
    main = LoadImage("Main.jpg")
    finish = LoadImage("End.jpg")
    credits = LoadImage("Credits.jpg")
    ring = LoadImage("Ring.png",0)
    button = LoadImage("Button.bmp",1)
    fail = LoadImage("Fail.jpg")
    rem *** Load sounds ***
    ringsound = LoadSound("Click.wav")
    rem *** Load music ***

```

```

    backgroundmusic = LoadMusic("Background.wav")
endfunction

rem *** Set up main section of the game ***
function SetUpGameScreen()
    rem *** Play music ***
    PlayMusic(backgroundmusic)
    rem *** Show main screen ***
    CreateSprite(1,main)
    SetSpriteSize(1,100,100)
    rem *** Load rings at image differences ***
    CreateSprite(2,ring)
    SetSpriteSize(2,-1,10)
    SetSpritePosition(2,91,86)
    CloneSprite(3,2)
    SetSpritePosition(3,51.5,22)
    CloneSprite(4,2)
    SetSpritePosition(4,49,68)
    CloneSprite(5,2)
    SetSpritePosition(5,73,66)
    CloneSprite(6,2)
    SetSpritePosition(6,88.5,66)
    CloneSprite(7,2)
    SetSpritePosition(7,55.75,62.5)
    rem *** Hide rings ***
    for c = 2 to 7
        SetSpriteDepth(c,9)
        SetSpriteVisible(c,0)
    next c
    rem *** Update screen ***
    Sync()
endfunction

function PlayGame()
    Print("PlayGame")
endfunction 10

function EndGame(timetaken)
    Print("EndGame")
endfunction

```

The new function should begin by starting the background music. However, it fails when attempting to do this because it does not recognise the ID given for the music resource.

Activity 8.21

Code for the updated version of *SpotTheDifference2*:

```

rem *** Define global variables ***
rem *** IDs for images ***
global main, finish, credits,ring, button, fail
rem *** IDs for sound ***
global ringsound
rem *** IDs for music ***
global backgroundmusic

LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end

rem *** Load resources ***
function LoadResources()
    rem *** Load images ***
    main = LoadImage("Main.jpg")
    finish = LoadImage("End.jpg")
    credits = LoadImage("Credits.jpg")
    ring = LoadImage("Ring.png",0)
    button = LoadImage("Button.bmp",1)
    fail = LoadImage("Fail.jpg")
    rem *** Load sounds ***
    ringsound = LoadSound("Click.wav")
    rem *** Load music ***
    backgroundmusic = LoadMusic("Background.wav")
endfunction

rem *** Set up main section of the game ***
function SetUpGameScreen()
    rem *** Play music ***
    PlayMusic(backgroundmusic)
    rem *** Show main screen ***
    CreateSprite(1,main)
    SetSpriteSize(1,100,100)
    rem *** Load rings at image differences ***

```



```

CreateSprite(2,ring)
SetSpriteSize(2,-1,10)
SetSpritePosition(2,91,86)
CloneSprite(3,2)
SetSpritePosition(3,51.5,22)
CloneSprite(4,2)
SetSpritePosition(4,49,68)
CloneSprite(5,2)
SetSpritePosition(5,73,66)
CloneSprite(6,2)
SetSpritePosition(6,88.5,66)
CloneSprite(7,2)
SetSpritePosition(7,55.75,62.5)
rem *** Hide rings ***
for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
next c
rem *** Update screen ***
Sync()
endfunction

function PlayGame()
    Print("PlayGame")
endfunction 10

function EndGame(timetaken)
    Print("EndGame")
endfunction

```

The program operates correctly now, getting as far as showing the main game screen.

Activity 8.22

Code for the updated version of *SpotTheDifference2*:

```

rem *** Define global variables ***
rem *** IDs for images ***
global main, finish, credits,ring, button, fail
rem *** IDs for sound ***
global ringsound
rem *** IDs for music ***
global backgroundmusic

LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end

rem *** Load resources ***
function LoadResources()
    rem *** Load images ***
    main = LoadImage("Main.jpg")
    finish = LoadImage("End.jpg")
    credits = LoadImage("Credits.jpg")
    ring = LoadImage("Ring.png",0)
    button = LoadImage("Button.bmp",1)
    fail = LoadImage("Fail.jpg")
    rem *** Load sounds ***
    ringsound = LoadSound("Click.wav")
    rem *** Load music ***
    backgroundmusic = LoadMusic("Background.wav")
endfunction

rem *** Set up main section of the game ***
function SetUpGameScreen()
    rem *** Play music ***
    PlayMusic(backgroundmusic)
    rem *** Show main screen ***
    CreateSprite(1,main)
    SetSpriteSize(1,100,100)
    rem *** Load rings at image differences ***
    CreateSprite(2,ring)
    SetSpriteSize(2,-1,10)
    SetSpritePosition(2,91,86)
    CloneSprite(3,2)
    SetSpritePosition(3,51.5,22)
    CloneSprite(4,2)
    SetSpritePosition(4,49,68)
    CloneSprite(5,2)
    SetSpritePosition(5,73,66)
    CloneSprite(6,2)
    SetSpritePosition(6,88.5,66)
    CloneSprite(7,2)

```

```

SetSpritePosition(7,55.75,62.5)
rem *** Hide rings ***
for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
next c
rem *** Update screen ***
Sync()
endfunction

rem *** Play game ***
function PlayGame()
    rem *** Start timer ***
    start = GetSeconds()
    CreateText(1,str(timetaken))
    SetTextColor(1,0,0,255)
    SetTextPosition(1,88,6)
    rem *** Set count of differences found ***
    found = 0
    rem *** Number of clicks so far is zero ***
    rem *** Get user clicks until all 6 differences
    found ***
    repeat
        rem *** Check for clicked(pressed)
        pressed = GetPointerPressed()
        rem *** IF pressed, ***
        if pressed = 1
            rem *** Add 1 to clicks ***
            inc presscount
            rem *** Check for sprite hit ***
            x = GetPointerX()
            y = GetPointerY()
            hit = GetSpriteHit(x,y)
            rem *** IF clicked over hidden ring THEN
            if hit > 1 and hit <=7 and
                GetSpriteVisible(hit) = 0
                rem *** Show ring ***
                SetSpriteVisible(hit,1)
                rem *** Play sound effect ***
                PlaySound(ringsound)
                rem *** Add 1 to differences found ***
                found = found + 1
            endif
        endif
        rem *** Update time so far ***
        timetaken = GetSeconds() - start
        SetTextString(1,Str(timetaken))
        Sync()
        until found = 6 or presscount = 7
    rem *** Delete existing sprites ***
    for c = 1 to 7
        DeleteSprite(c)
    next c
    rem *** Delete sound ***
    DeleteSound(ringsound)
    rem *** Delete text ***
    DeleteText(1)
endfunction timetaken

function EndGame(timetaken)
    Print("EndGame")
endfunction

```

The program now allows the player to click on the six differences.

Activity 8.23

Code for the updated version of *SpotTheDifference2*:

```

rem *** Define global variables ***
rem *** IDs for images ***
global main, finish, credits,ring, button, fail
rem *** IDs for sound ***
global ringsound
rem *** IDs for music ***
global backgroundmusic

LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
end

rem *** Load resources ***
function LoadResources()
    rem *** Load images ***
    main = LoadImage("Main.jpg")

```

```

finish = LoadImage("End.jpg")
credits = LoadImage("Credits.jpg")
ring = LoadImage("Ring.png",0)
button = LoadImage("Button.bmp",1)
fail = LoadImage("Fail.jpg")
rem *** Load sounds ***
ringsound = LoadSound("Click.wav")
rem *** Load music ***
backgroundmusic = LoadMusic("Background.wav")
endfunction

rem *** Set up main section of the game ***
function SetUpGameScreen()
    rem *** Play music ***
    PlayMusic(backgroundmusic)
    rem *** Show main screen ***
    CreateSprite(1,main)
    SetSpriteSize(1,100,100)
    rem *** Load rings at image differences ***
    CreateSprite(2,ring)
    SetSpriteSize(2,-1,10)
    SetSpritePosition(2,91,86)
    CloneSprite(3,2)
    SetSpritePosition(3,51.5,22)
    CloneSprite(4,2)
    SetSpritePosition(4,49,68)
    CloneSprite(5,2)
    SetSpritePosition(5,73,66)
    CloneSprite(6,2)
    SetSpritePosition(6,88.5,66)
    CloneSprite(7,2)
    SetSpritePosition(7,55.75,62.5)
    rem *** Hide rings ***
    for c = 2 to 7
        SetSpriteDepth(c,9)
        SetSpriteVisible(c,0)
    next c
    rem *** Update screen ***
    Sync()
endfunction

rem *** Play game ***
function PlayGame()
    rem *** Reset aspect ratio ***
    SetDisplayAspect(768/1024.0)
    rem *** Start timer ***
    start = GetSeconds()
    CreateText(1,str(timetaken))
    SetTextColor(1,0,0,255)
    SetTextPosition(1,88,6)
    rem *** Set count of differences found ***
    found = 0
    rem *** Number of clicks so far is zero ***
    rem *** Get user clicks until all 6 differences
    found ***
    repeat
        rem *** Check for clicked(pressed)
        pressed = GetPointerPressed()
        rem *** IF pressed, ***
        if pressed = 1
            rem *** Add 1 to clicks ***
            inc presscount
            rem *** Check for sprite hit ***
            x = GetPointerX()
            y = GetPointerY()
            hit = GetSpriteHit(x,y)
            rem *** IF clicked over hidden ring THEN
            if hit > 1 and hit <=7 and
            GetSpriteVisible(hit) = 0
                rem *** Show ring ***
                SetSpriteVisible(hit,1)
                rem *** Play sound effect ***
                PlaySound(ringsound)
                rem *** Add 1 to differences found ***
                found = found + 1
            endif
        endif
        rem *** Update time so far ***
        timetaken = GetSeconds() - start
        SetTextString(1,Str(timetaken))
        Sync()
    until found = 6 or presscount = 7
    rem *** Delete existing sprites ***
    for c = 1 to 7
        DeleteSprite(c)
    next c
    rem *** Delete sound ***
    DeleteSound(ringsound)
    rem *** Delete text ***
    DeleteText(1)

```

```

endfunction timetaken

rem *** Finish game ***
function EndGame(timetaken)
    rem *** Wait before finishing ***
    Sleep(1000)
    rem *** Reset aspect ratio ***
    SetDisplayAspect(768.0/1024.0)
    rem *** IF all differences found ***
    if found = 6
        rem *** Show End screen... ***
        CreateSprite(1,finish)
        SetSpriteSize(1,100,100)
        rem *** ...and total time taken ***
        CreateText(1,str(timetaken))
        SetTextColor(1,0,0,255)
        SetTextPosition(1,36,31)
        Sync()
    else
        rem *** Show Fail screen... ***
        CreateSprite(1,fail)
        SetSpriteSize(1,100,100)
    endif
    rem *** ... with button... ***
    CreateSprite(2,button)
    SetSpriteSize(2,15,-1)
    SetSpritePosition(2,80,90)
    rem *** Allow for Credits button press ***
    do
        pressed = GetPointerPressed()
        if pressed = 1
            x = GetPointerX()
            y = GetPointerY()
            hit = GetSpriteHit(x,y)
            rem *** IF Credit button pressed THEN ***
            if hit =2
                rem *** Show credits for 5 secs ***
                CreateSprite(3,credits)
                SetSpriteSize(3,100,100)
                SetSpriteDepth(3,8)
                Sync()
                Sleep(5000)
                rem *** Remove Credits screen ***
                DeleteSprite(3)
            endif
        endif
        Sync()
    loop
endfunction

```

Even when we find all 6 differences the game shows the *Fail* screen.

The variable required is *found* which contains the count of how many differences were found by the player.

We need to add the code

```

rem *** Differences found ***
global found

```

The program works correctly after these lines have been added.

Activity 8.24

Although the app runs, it is not in landscape mode for the main screen.

Activity 8.25

The modified version of *SetUpGameScreen()*:

```

rem *** Set up main section of the game ***
function SetUpGameScreen()
    rem *** Set screen aspect ***
    SetDisplayAspect(1024.0/768)
    rem *** Play music ***
    PlayMusic(backgroundmusic)
    rem *** Show main screen ***
    CreateSprite(1,main)
    SetSpriteSize(1,100,100)
    rem *** Load rings at image differences ***
    CreateSprite(2,ring)
    SetSpriteSize(2,-1,10)

```

```

SetSpritePosition(2,91,86)
CloneSprite(3,2)
SetSpritePosition(3,51.5,22)
CloneSprite(4,2)
SetSpritePosition(4,49,68)
CloneSprite(5,2)
SetSpritePosition(5,73,66)
CloneSprite(6,2)
SetSpritePosition(6,88.5,66)
CloneSprite(7,2)
SetSpritePosition(7,55.75,62.5)
rem *** Hide rings ***
for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
next c
rem *** Update screen ***
Sync()
endfunction

```

The game should now play correctly on your device.

Activity 8.26

FUNCTION NAME	:	SetUpGameScreen
PARAMETERS		
In	:	None
Out	:	None
GLOBALS		
Read	:	backgroundmusic, main, ring
Written	:	None
PRE-CONDITION	:	None
DESCRIPTION	:	Sets the aspect ratio of the screen to landscape, starts the background music playing, displays the main game screen, positions the rings over the image differences and hides the rings.

FUNCTION NAME	:	PlayGame
PARAMETERS		
In	:	None
Out	:	timetaken : integer
GLOBALS		
Read	:	ringsound
Written	:	found
PRE-CONDITION	:	None
DESCRIPTION	:	Allows the player to click on the screen up to 7 times. Keeps a count of how many have been found. Stops when all 6 found or when 7 clicks made.

FUNCTION NAME	:	EndGame
PARAMETERS		
In	:	timetaken : integer
Out	:	None
GLOBALS		
Read	:	found, finish, fail
Written	:	None
PRE-CONDITION	:	None
DESCRIPTION	:	The display returns to portrait layout. If all 6 differences found, the routine displays the Finish screen and the time taken in seconds. If all 6 are not found, the fail screen is displayed. If the Credits button is pressed, the Credits screen shows for 5 seconds.

Activity 8.27

To check that the *Fail* screen appears correctly, modify the line

```
found = 6
```

in the main section of the program to read

```
found = 5
```


String and Math Functions

In this Chapter:

- ☐ Standard String-Handling Functions
- ☐ Adding New String-Handling Functions
- ☐ Updating the StringLibrary File
- ☐ Understanding Cartesian Coordinates
- ☐ Math Functions
- ☐ Using `sin()` and `cos()` to Calculate Coordinates

String Functions

Introduction

Unlike numeric variables which hold only a single value, strings can hold a whole collection of characters, perhaps several words or even sentences. For example, it's quite valid to store a line of text in a string variable with a statement such as:

```
poem$ = "Mary had a little lamb"
```

Because a string can contain so many characters, there are several operations that programmers find themselves needing to do with strings. For example, we might want to find out how many characters are in a string, convert a string to uppercase, or extract part of a string.

AGK BASIC contains a number of string-handling functions as part of the core language. These functions are listed and explained in the first part of this chapter.

String-Handling Functions

Len()

The `Len()` function returns the number of characters in a string. The string to be examined is given in parentheses. For example, the expression

```
Len("Hello")
```

would return the value 5 since there are 5 characters in the word *Hello*.

Spaces and any other non-alphabetic symbols within a string also count as characters, so the line

```
Len("Hello, world?")
```

would return the value 13, since it will include the comma, space and question mark in the count.

FIG-9.1

Len()

The `Len()` function has the format shown in FIG-9.1.

integer   string 

where :

string is a string constant, string variable, or string expression.

As with any function that returns a value, this value can be displayed, assigned to a variable, or used in an expression. Hence each of the following lines are valid:

<code>Print(Len("Hello"))</code>	<code>`displays 5</code>
<code>result = Len("Hello")</code>	<code>`sets result equal to 5</code>
<code>ans = Len("Hello") * 3</code>	<code>`sets ans to 15 (5 x 3)</code>
<code>if Len("Hello") > 3</code>	<code>`condition is true since 5 > 3</code>

Of course, it's much more likely that you'll use a string variable as an argument rather than a string constant.

Activity 9.1

In this Activity we are going to make use of our StringLibrary.agc file which we placed in the Library folder in the last chapter.

Start a new project called *TestLen*. Compile the default code.

Using Windows Explorer, make a copy of the *StringLibrary.agc* file found at *HandsOnAGK/Library* and paste it into the *TestLen* folder.

Modify the contents of *main.agc* to read:

```
rem *** Test Len() Function ***
rem *** Include Library function ***
#include "StringLibrary.agc"

rem *** Generate string ***
text$ = RandomString(-1)
rem *** Print string and its length ***
Print(text$)
Print(Len(text$))
Sync()
do
loop
```

Test and save your program.

Notice that in order to use the *RandomString()* function, it is necessary to add a

```
#include "StringLibrary.agc"
```

command at the start of the program.

Upper()

The `Upper()` function takes a string argument and returns the uppercase version of that string. For example, the line

```
Print(Upper("Hello"))
```

would display the word *HELLO*.

Any characters in the string that are not letters are returned unchanged by this statement. Hence,

```
Print(Upper("Abc123"))
```

would display *ABC123*.

Activity 9.2

What would be the value of *b\$* after the following lines are executed?

```
a$ = "1-by-1"
b$ = Upper(a$)
```

FIG-9.2

Upper()

The `Upper()` statement has the format shown in FIG-9.2.

string `Upper` `(` string `)`

where:

string is any string value.

Lower()

The `Lower()` function takes a string argument and returns the lowercase version of that string. Any non-alphabetic characters in the string are returned unchanged.

```
Print(Lower("Hello"))
```

would display the word *hello*.

FIG-9.3

Lower()

This statement has the format shown in FIG-9.3.

string `Lower` `(` string `)`

where:

string is any string value.

Left()

It's possible to extract the left-hand section of a string using the `Left()` function. This time you need to include two parameters: the first is the string itself, and the second is the number of characters you want to extract. For example,

```
Print(Left("abcdef",2))
```

would display *ab* on the screen, `Left()` having returned the left two characters from the string *abcdef*.

If the number given is larger than the number of characters in the string as in

```
ans$ = Left("abcdef",10)
```

then the complete string is returned (i.e. *abcdef*)

Should a zero, or negative value be given as in

```
result$ = Left("abcdef",0)
```

then the returned string contains no characters. That is, *result\$* will hold an empty string.

FIG-9.4

Left()

The `Left()` function has the format shown in FIG-9.4.

string `Left` `(` string `,` inum `)`

where:

string is any string value.

inum is a positive integer value giving the number of characters to be copied. It should be in the range 0 to the number of characters in the string.

Right()

If we want to extract the right-hand part of a string we can use the `Right()` function. For example, the statement

```
Print(Right("abcdef",2))
```

would display *ef* on the screen.

The statement has the format shown in FIG-9.5.

FIG-9.5

Right()

string `Right` (`(` string `,` inum `)`

where:

string is any string value.

inum is a positive integer value giving the number of characters to be copied. It should be in the range 0 to the number of characters in the string.

Mid()

This statement extracts a substring from the specified string. The position of the first character and the number of characters to be extracted is given as the second and third arguments to the function. For example, the statement

```
letter$ = Mid("abcdef",4,2)
```

would place the value *de* in *letter\$* (extracts 2 characters starting at the 4th character in the string). We can use this statement to access each character in a string. For example, the code snippet

```
text$ = RandomString(-1)
for c = 1 to Len(text$)
    Print(Mid(text$,c,1))
next c
Sync()
```

will display each character of the string stored in `text$` on a separate line.

Activity 9.3

Create a new project, *Letters*, which makes use of the code above to display a generated string and then displays the individual characters of the string. Remember to copy the *StringLibrary.agc* file into the project folder and add a `#include` instruction to your code.

Modify the program so that the characters are displayed in reverse order on a single line.

Change the program so that, rather than display the characters, it counts how many E's are in the string.

FIG-9.6

Mid()

The format for the **Mid()** statement is given in FIG-9.6.

string **Mid** ((string , ipost , inum)

where:

- string** is any string value.
- ipost** is a positive integer giving the position of the first character to be extracted. Range 1 to length of string.
- inum** is a positive integer giving the number of characters to be copied.

Asc()

ASCII character codes are given in Appendix A at the end of the book.

This function returns an integer value representing the ASCII value of the first character in the string supplied. A typical statement such as

```
Print(Asc("ABC"))
```

would display the value 65 since that is the ASCII code for a capital A. Using this function on an empty string as in the line

```
result = Asc("")
```

returns the value zero.

FIG-9.7

Asc()

The format for this statement is given in FIG-9.7.

integer **Asc** ((string)

where:

- string** is any string value, but only the first character is used by the function.

Chr()

The **Chr()** function complements the **Asc()** function by returning the character whose ASCII code matches the specified value. For example, the line

```
Print(Chr(65))
```

would display a capital letter A since the ASCII code for a capital A is 65.

ASCII 32 is the space character. So although it is displayable, there's not much to see!

The value given should lie between 0 and 127. However, only characters with an ASCII code of 32 to 126 are displayable; other values are used for various control purposes and attempting to display such values has no visible effect in AGK BASIC.

We could display all the letters of the alphabet in uppercase using the lines:

```
for c = 1 to 26
  Print(Chr(64+c))
next c
Sync()
```

FIG-9.8

Chr()

The format for this statement is given in FIG-9.8.

string **Chr** (**ival**)

where:

ival is an integer value. This value must be between 0 and 127, but is more likely to be between 32 and 126, these being the ASCII range of values for all displayable characters.

Activity 9.4

Create a new project called *ASCIITable*.

Code the program so that it displays the numbers 32 to 126 and, beside each number, the corresponding ASCII character.

Get the program to pause after every 25 characters, waiting for 5 seconds before continuing.

Test and save your program.

Str()

The **str()** function takes a numeric argument and returns a string containing the same digits as the argument. For example, the line

```
result$ = Str(123)
```

will store the string *123* in the variable *result\$*

When converting a real value, the number of decimal places required can be specified, as in the line

```
value$ = Str(12.326,2)
```

which will store *12.33* in *value\$*. Note that the last digit is rounded.

Perhaps the most useful application of this function is to simplify output involving several values. For example, in past programs we have had to write code such as:

```
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
```

Using the **str()** function we can now rewrite this as:

```
Print("My number was : " + Str(dice))
Print("Your guess was : " + Str(guess))
```

The + operator is used to join two strings.

FIG-9.9

Str()

This statement has the format shown in FIG-9.9.

string **Str** (**value** [**, iplaces**])

where:

value is any numeric value.

iplaces is an integer value giving the number of decimal places to be stored.

Activity 9.5

Start a new project called *CountZero* and write a program to generate a random number between 1000 and 65000 which displays the number generated and a count of how many zeros are in that number.

(HINT: Convert the number to a string then count the number of zeros in that string.)

Test and save your project.

Val()

This function takes a string argument and returns the integer equivalent. The string should contain only numeric characters. For example, executing the line

```
result = Val("123")
```

will store the value 123 in the variable *result*.

If the string contains a real value, only the integral part will be converted. So the line

```
ans# = Val("123.45")
```

gives *ans#* a value of 123.0 when displayed.

If the string contains a mixture of numeric and non-numeric characters, the value returned is constructed from all numeric characters preceding the first non-numeric character. For example, the call

```
Val("12ABC3")
```

returns the value 12.

If the string starts with a non-numeric character (other than a sign or decimal point) then the function returns zero.

The string may hold a value which represents a number in a different number base. But when the value is not a base 10 number, we need to add a second parameter giving the number's base. So for example, we could convert a string representing a hexadecimal number using the line:

```
num = Val("FE", 16)
```

which would store the value 254 (the decimal equivalent of FE_{16}) in *num*.

Although the most obvious number bases for a computer system are 2 (binary), 16 (hexadecimal) and 8 (octal), you can have any integer base you wish. For example, the statement

```
v = Val("210", 3)
```

states that 210 is a base 3 number and therefore *v* will be set to 21 ($2*9+1*3+0*1$).

FIG-9.10

Val()

The `Val()` function has the format shown in FIG-9.10.

integer `Val` `(` string `[` , `ibase` `]` `)`

where:

string is a string containing only numeric characters, a decimal point, or a sign (+ or -).

ibase is an positive integer value giving the number base.

ValFloat()

FIG-9.11

ValFloat()

To convert a string to a real number, use `ValFloat()` (see FIG-9.11).

float `ValFloat` `(` string `)`

where:

string is a string containing only numeric characters, a decimal point, or a sign (+ or -).

Space()

Although it is easy enough to create a string full of spaces with a line such as

```
text$ = "          "
```

if you want an exact number of spaces in your string, then it's easier to use the `Space()` function which returns a string containing a specified number of spaces.

```
text$ = Space(23)
```

assigns a string containing 23 spaces to the variable `text$`. The format for this statement is shown in FIG-9.12.

FIG-9.12

Space()

string `Space` `(` `ivalue` `)`

where:

ivalue is a positive integer which specifies the number of spaces in the string returned by the function.

Bin()

As you know, the computer uses the binary number system when storing programs and data. If you'd like to see what a specific integer value looks like in binary, this function will do the job for you. It returns a string showing the binary representation of a specified integer value. For example, the instruction

```
binary$ = Bin(65)
```

would assign the string `1000001` (the binary equivalent of 65) to the variable `binary$`.

`Print(Bin(-65))`

would display the string `IIIIIIIIIIIIIIIIIIII0IIIII`. The format for the `Bin()` function is shown in FIG-9.13.

Bin()

where:

Hex()

```
hexadecimal$ = Hex(65)
```

For negative values, the hexadecimal string returned is the equivalent of the 2's complement form. Therefore,

```
Print(Hex(-15))
```

Hex()

where:

Activity 9.6

Use the Button functions to read in an integer value and then display the equivalent binary and hexadecimal value. Test and save your project.

CountStringTokens()

It doesn't matter what characters make up a token, nor which character is used as a delimiter. In fact, you can use several different delimiters in the same string.

FIG-9.15

CountStringTokens()

integer **CountStringTokens** ((string , sdelimits))

where

string is a string containing the characters to be processed.**sdelimits** is a string giving the delimiters to be assumed when identifying the tokens.

For example, the statement

```
Print(CountStringTokens("red,green,blue,yellow,white",","))
```

will display the value 5.

The line

```
Print(CountStringTokens("1/2:3|4","/:|"))
```

will display the value 4. In this case any of the characters / : or | are taken as delimiters.

GetStringToken()

To extract the identified tokens from a string we can use the **GetStringToken()** statement (see FIG-9.16).**FIG-9.16**

GetStringToken()

string **GetStringToken** ((string , sdelimits , indx))

where

string is a string containing the characters to be processed.**sdelimits** is a string giving the delimiters to be assumed when identifying the tokens.**indx** is an integer giving the number of the token to be returned (the first token is at position 1).

For example, the line

```
Print(GetStringToken("red/green/blue/yellow", "/", 3))
```

would display the term *blue* (the third token in the string).

The program in FIG-9.17 displays the number of tokens in a string and then lists them separately.

FIG-9.17Using the *StringToken*
Functions

```
rem *** Using Tokens ***
```

```
rem *** Set string and delimiters ***
```

```
quote$ = "It is a truth universally acknowledged, that a single  
man in possession of a good fortune, must be in want of a wife"  
delimiters$=" , " //Space and comma
```



FIG-9.17

(continued)

Using the *StringToken*
Functions

```

rem *** Get and display token count ***
tokens = CountStringTokens(quote$,delimiters$)
Print(tokens)

rem *** Display each token ***
for c = 1 to tokens
    Print(GetStringToken(quote$,delimiters$,c))
next c
Sync()
do
loop

```

Activity 9.7

Start a new project called *Tokens* and implement the code given in FIG-9.17.

Test your code. Test the program again with a quote and delimiters with options of your own. Save your project.

Creating Your Own String Functions

There are several more operations which would be useful to have when manipulating strings, and, although AGK BASIC does not contain commands to perform these operations, we can easily write them ourselves. Some of these are described below.

Pos()

The *Pos()* function returns the position of a specified character in a specified string. For example, the line

```
place = Pos("abcd","c")
```

would assign the value 3 to *place*, since *c* occurs at position 3 in the string *abcd*.

If the character being searched for occurs more than once in the string, then it is the position of the first occurrence that is returned. Hence, the call

```
Pos("abdc", "c")
```

would return the value 3, not 5. If the character being searched for does not occur within the string, then a value of 0 is returned. The mini-spec for this function is:

FUNCTION NAME	:	Pos
PARAMETERS		
In	:	s : string f : character
Out	:	result : integer
PRE-CONDITION	:	None
DESCRIPTION	:	<i>result</i> is set to the position at which <i>f</i> first occurs in <i>s</i> . If <i>f</i> does not occur in <i>s</i> , then <i>result</i> is set to zero.

The code for this function is shown in FIG-9.18.

FIG-9.18

The Pos() Function's
Code

```
rem *** Find Position of character in string ***  
  
function Pos(s$, f$)  
    rem *** result stays at 0 if no match found ***  
    result = 0  
    rem *** Make sure we're looking for a single character ***  
    first$ = Mid(f$,1,1)  
    rem *** FOR each character in s$ DO ***  
    for c = 1 to Len(s$)  
        rem *** IF that character matches what we're after THEN ***  
        if Mid(s$,c,1) = first$  
            rem *** Set result to this position and exit loop ***  
            result = c  
            exit  
        endif  
    next c  
endfunction result
```

Because AGK BASIC allows only string variables and not single character ones (as some other languages offer), we cannot be sure that when the function *Pos()* is called, the second argument, *f\$*, contains only a single character. For example, the line

```
result = Pos("abcdef", "ei")
```

would be valid, even though there is more than one character in the second parameter. But by including the line

```
first$ = Mid(f$,1,1)
```

in the code for *Pos()*, we extract the first character from *f\$*. It is this first character that we then search for in *s\$*.

Activity 9.8

Start a new project called *FunctionTester*.

Copy the file *StringLibrary.agc* from the *Library* folder into the new project's folder.

In *main.agc*, add the code for function *Pos()* as given in FIG-9.18.

In the main part of the program, create a random string 30 characters in length and use a call to *Pos()* to display the first occurrence of a capital *D* within the random string. The generated string should also be displayed so you can check that the result from *Pos()* is correct.

Check that *Pos()* also works when the character searched for cannot be found.

Save your project.

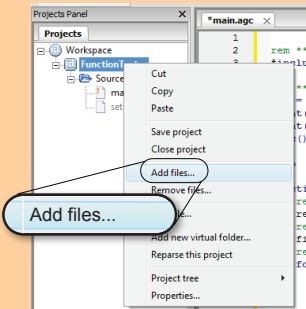
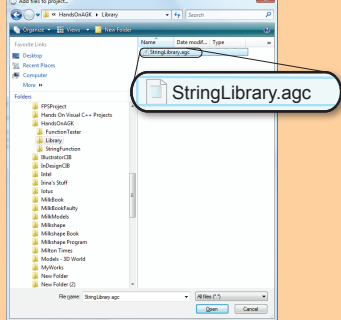
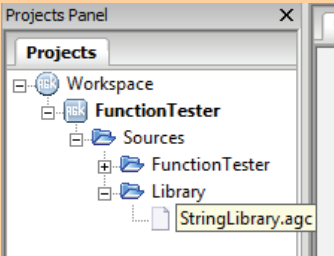
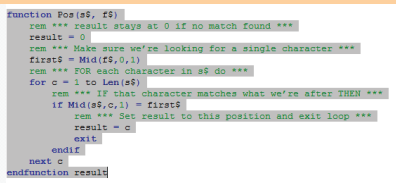
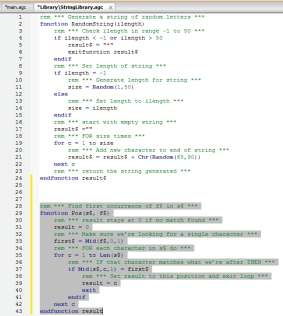
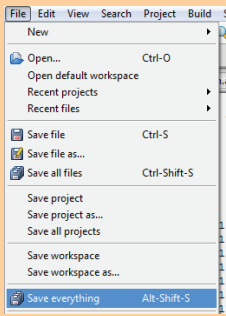
Pos() is another function that could prove useful in later projects, so it will be worth adding its code to the *StringLibrary.agc* file in the *Library* folder.

The only thing we have to watch out for here is that we paste the code into the original *StringLibrary.agc* file held in the *Library* folder. FIG-9.19 shows how to

update the original *StringLibrary.agc* file.

FIG-9.19

Adding a New Function to *StringLibrary.agc*

<p>In the Projects Panel, right-click on the project name and select Add files from the pop-up menu.</p>	<p>Next, select <i>StringLibrary.agc</i> from the <i>Library</i> folder.</p>
	
<p>The file is now included in the <i>Sources</i> section of the project.</p>	<p>From <i>main.agc</i>, we copy the code for <i>Pos()</i>.</p>
	
<p>Double clicking on <i>StringLibrary</i> in the <i>Projects Panel</i> will open the file in the edit area and we can paste the code for <i>Pos()</i> to the file.</p>	<p>Finally, selecting File Save everything from the main menu will save the updated <i>StringLibrary</i> file.</p>
	

Activity 9.9

Update the contents of the *StringLibrary.agc* file by adding the *Pos()* function as described in FIG-9.19.

Occurs()

The *Occurs()* function returns how often a specified character appears within a

specified string. Hence, the expression

```
Occurs("abcde","c")
```

would return 2 since *c* occurs twice within *abcde*. The mini-spec for the routine is:

FUNCTION NAME	:	Occurs
PARAMETERS		
In	:	s : string
f	:	character
Out	:	result : integer
PRE-CONDITION	:	None
DESCRIPTION	:	<i>result</i> is set to the number of times <i>f</i> occurs in <i>s</i> .

The code for this function is shown in FIG-9.20.

FIG-9.20

The Occurs()
Function's Code

```
rem *** Return how often f$ occurs in s$ ***
function Occurs(s$,f$)
    rem *** None found so far ***
    result = 0
    rem *** Make sure only one character ***
    first$ = Mid(f$,0,1)
    rem *** FOR each character in s$ Do ***
    for c = 1 to Len(s$)
        rem *** if it matches req'd character, add 1 to result ***
        if Mid(s$,c,1) = first$
            result = result + 1
        endif
    next c
endfunction result
```

Activity 9.10

Add the code for *Occurs()* to *main.agc* in *FunctionTester*.

In the main part of the program, create a random string 30 characters in length and use a call to *Occurs()* to display how often a capital *S* appears within the random string. The generated string should also be displayed so you can check that the result from *Occurs()* is correct.

Save your project. Add the code for *Occurs()* to *StringLibrary.agc* in the *Library* folder.

Insert()

The **Insert()** function returns a string created by inserting one string into another, starting at a specified position. For example, the line

```
Print(Insert("abcdef ","xy", 4))
```

would display the string *abcxydef* having inserted the string *xy* into string *abcdef* starting at position 4.

If an attempt is made to insert the second string at an invalid position, then the

returned string is an exact match of the first string.

The function's mini-spec is:

FUNCTION NAME	:	Insert
PARAMETERS		
In	:	s : string f : string p : integer
Out	:	result : string
PRE-CONDITION	:	None
DESCRIPTION	:	<i>result</i> is created by inserting <i>f</i> into <i>s</i> at position <i>p</i> . Normally, <i>p</i> should be in the range 1 to Len(<i>s</i>)+1. If <i>p</i> is outside this range result is set equal to <i>s</i> .

The code for this routine is given in FIG-9.21.

FIG-9.21

The Insert() Function's
Code

```
rem *** Returns string with f$ inserted at position p into s$ ***  
  
function Insert(s$,f$,p)  
    rem *** If invalid position, result is original string ***  
    if p < 1 or p > Len(s$)+1  
        result$ = s$  
    else  
        rem *** split s$ into two parts & insert f$ in between ***  
        result$ = Left(s$,p-1)  
        result$ = result$ + f$  
        result$ = result$+ Right(s$,Len(s$)-(p-1))  
    endif  
endfunction result$
```

Notice that the main logic in the function involves splitting the first string into two parts and inserting the second string in between these parts.

Activity 9.11

Add the code for *Insert()* to *main.agc* in *FunctionTester*.

In the main part of the program, call *Insert()* to add *XX* to *ABCDEFGHI* starting at position 2.

Test and save your project.

Also check that the function performs as specified if the insert position given is invalid.

Add the code for *Insert()* to *StringLibrary.agc* in the *Library* folder.

Delete()

The *Delete()* function returns a string created by deleting a specified section of an original string. For example, the line

```
temp$ = Delete("abcdefghi",2,4)
```

would set *temp\$* to *afghi* this being created by removing 4 characters, starting at position 2, from the original string *abcdefghi*.

If the start position is invalid, a copy of the original string is returned. If the number of characters to be deleted is too large, then as many characters as possible are removed.

The function's mini-spec is:

FUNCTION NAME	:	Delete
PARAMETERS		
In	:	s : string st : integer num : integer
Out	:	sresult : string
PRE-CONDITION	:	None
DESCRIPTION	:	<i>sresult</i> is equal to <i>s</i> with the <i>num</i> characters deleted starting from position <i>st</i> . If <i>st</i> is outside the range 1 to <i>Len(s)</i> , <i>sresult</i> is equal to <i>s</i> . If <i>num</i> > <i>Len(Right(s,Len(s)-st+1))</i> , <i>sresult</i> is set to <i>Left(s,st-1)</i> .

Notice how the mini-spec makes use of other string-handling functions to describe how the value of *result* is determined. Although more difficult to understand than plain English, this approach can often lead to a briefer description and will always be unambiguous.

The code for this routine is given in FIG-9.22.

FIG-9.22

The Delete() Function's Code

```
rem *** Returns string created by deleting num chars ***
rem *** from s$ starting at position st ***

function Delete(s$, st, num)
    rem *** if invalid position, result is original string ***
    if st < 1 or st > Len(s$)
        result$ = s$
    else
        rem *** Set result to the part of s$ to the left of ***
        rem *** the section to be deleted ***
        result$ = Left(s$, st-1)
        rem *** IF not deleting to the end of s$, ***
        rem *** add right section ***
        if st+num-1 <= Len(s$)
            result$ = result$+Right(s$,Len(s$)-(st+num-1))
        endif
    endif
endfunction result$
```

Activity 9.12

Add the code for *Delete()* to *main.agc* in *FunctionTester*.

In the main part of the program, call *Delete()* to delete from position 3 the next 5 characters. Use *ABCDEFGHI* as the string.

Test and save your project.

Also check that the function performs as specified if the start position given is invalid and when more characters than available are to be deleted.

Add the code for *Delete()* to *StringLibrary.agc* in the *Library* folder.

Replace()

The *Replace()* function is designed to return a string constructed by replacing a single character at a specified position in an original string. Therefore the line

```
ans$ = Replace$("abcdef", "x", 4)
```

sets *ans\$* equal to *abcxef* having replaced the fourth character in *abcdef* with an *x*.

If an invalid position is specified, then the original string is returned.

Activity 9.13

Create a mini-spec for the *Replace()* function.

Using the *FunctionTester* project, write code for the *Replace()* function and then test your coding.

Add the code for *Replace()* to *StringLibrary.agc* in the *Library* folder.

Summary

- The **Len()** function returns the number of characters in a specified string.
- The **Upper()** function returns the uppercase equivalent of a specified string.
- The **Lower()** function returns the lowercase equivalent of a specified string.
- The **Left()** function returns a left-hand sub-string from a specified string.
- The **Right()** function returns a right-hand sub-string from a specified string.
- The **Mid()** function returns a specified number of characters from a given position in a specified string.
- The **Asc()** function returns the ASCII code of a specified character.
- The **Chr()** function returns the character whose ASCII code matches a specified value.
- The **Str()** function returns the string equivalent of a specified number.
- The **Val()** function returns the numeric equivalent of a specified string.

- The `Space()` function returns a string containing a specified number of spaces.
- The `Bin()` function returns a string representing the binary equivalent of a specified integer.
- The `Hex()` function returns a string representing the hexadecimal equivalent of a specified integer.
- Use `CountStringTokens()` to count the number of tokens in a string.
- Use `GetStringToken()` to extract a specific token from a string.

Math Functions

Introduction

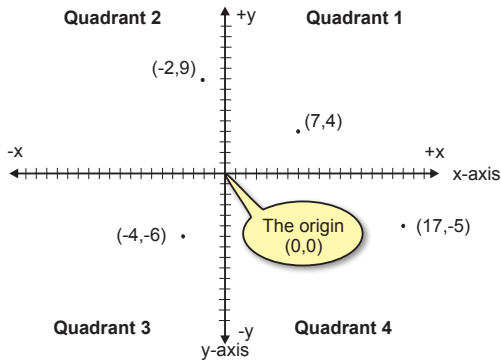
A second important group of standard programming functions is the math functions. All of the math functions not previously covered are given here.

Coordinates

In 2D coordinate geometry objects are positioned by specifying x,y **Cartesian coordinates** (see FIG-9.23).

FIG-9.23

Cartesian Coordinates



From FIG-9.22 we can see that the origin is the position where the two axes cross and that the axes split the area into four quadrants:

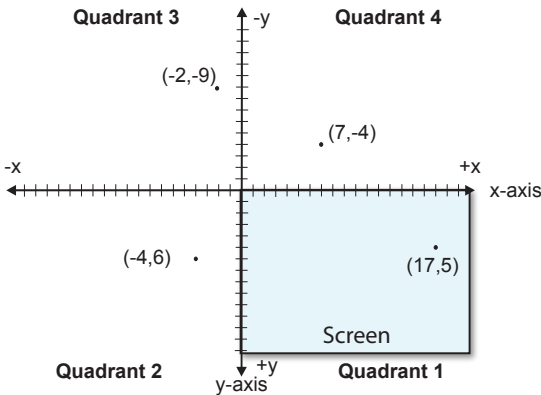
- quadrant 1: both x and y values are positive
- quadrant 2: x values are negative and y values positive
- quadrant 3: both x and y values are negative
- quadrant 4: x values are positive and y values negative

However, on a computer screen, the y axis has been turned upside down so that positive y values are at the bottom while negative y values are at the top (see FIG-9.24). Also, the top-left point on the screen is taken as the origin so a screen displays only quadrant 1 points.

FIG-9.24

Screen Coordinates

The area shown as the screen is not to scale.



This modification changes the position of the four quadrants. We'll be using this inverted coordinate system, since that's the one we need when creating games.

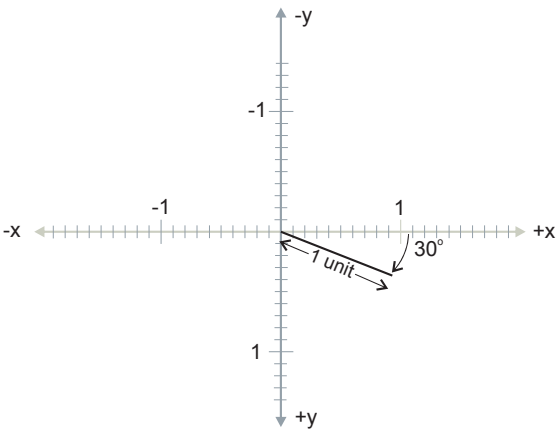
Trigonometric Functions

Cos()

If we draw a line starting at the origin which is exactly one unit in length at an angle of 30° to the x-axis, then we create the setup shown in FIG-9.25.

FIG-9.25

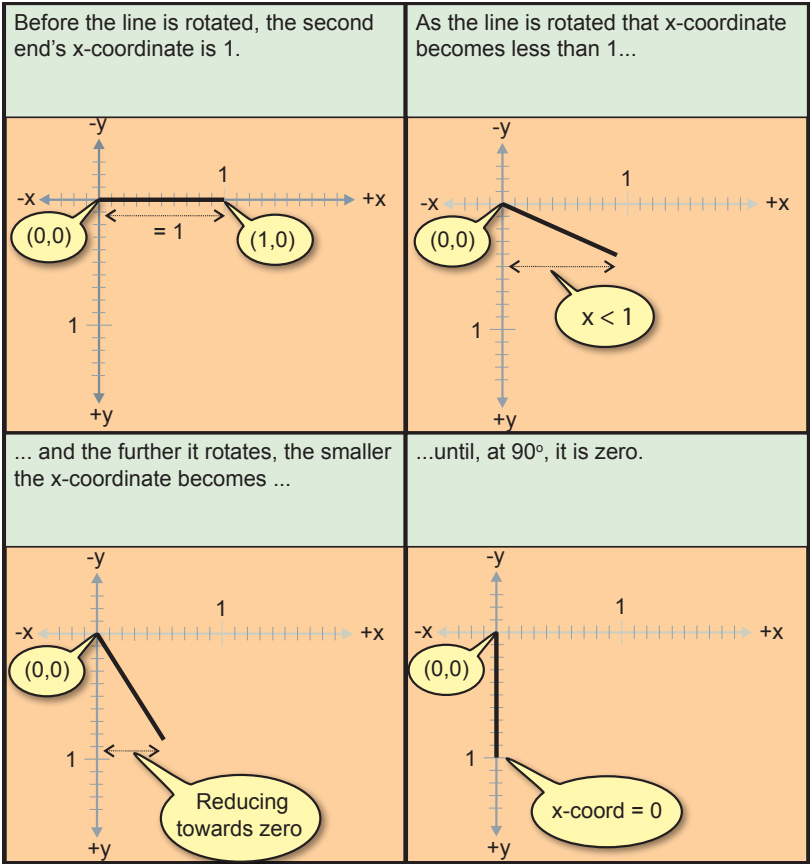
Measuring Angles



We know that one end of the line has the coordinates $(0,0)$, but what are the coordinates of the other end? We'll start by examining the x-coordinate. From FIG-9.26 we can see that the x-coordinate of the end point changes as we rotate the line to 70° from the x-axis and finally to 90° .

FIG-9.26

X-Coordinate
Determined by Angle



Activity 9.14

What would be the x-coordinate of the line if it was rotated to

- a) 0° b) 90°

Although it is easy enough to work out the x-coordinate when the line lies along one of the axes, things are a bit more difficult when some other angle of rotation is involved. Luckily for us, someone worked all the x-coordinates for every possible angle several hundred years ago and called it the **cosine** of the angle (often shortened to **cos**).

So, if we draw a line starting at the origin which is 1 unit in length and rotate it by an angle of theta (θ), then the x-coordinate for the other end of that line is given by the expression

$$\text{cosine}(\theta) \text{ or } \cos(\theta)$$

If we rotate our line by more than 90° it moves into quadrant 2 and the x-coordinate will become negative. As we pass 180° and move into quadrant 3, the x-coordinate remains negative, but after 270° , the x-coordinate is once again positive.

Activity 9.15

Load up Microsoft's *Calculator* program.

Choose **View|Scientific**. Make sure it is using decimal and degrees.

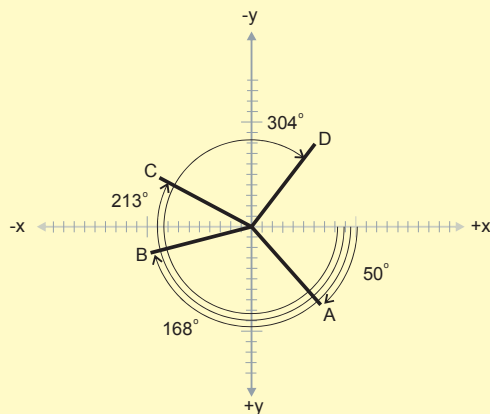
By calculating the cosine of the following angles (to 3 decimal places)

- a) 0° b) 90° c) 30° d) 70°

write down the x-coordinate of the lines of 1 unit which have been rotated by the angles given above.

Activity 9.16

By using the cosine function in *Calculator*, determine the x-coordinates of the lines shown below (all lines start at the origin and are 1 unit in length).



AGK BASIC performs this calculation using the `Cos ()` function which has the format shown in FIG-9.27.

FIG-9.27

`Cos()`

real `Cos` `(` `angle` `)`

where:

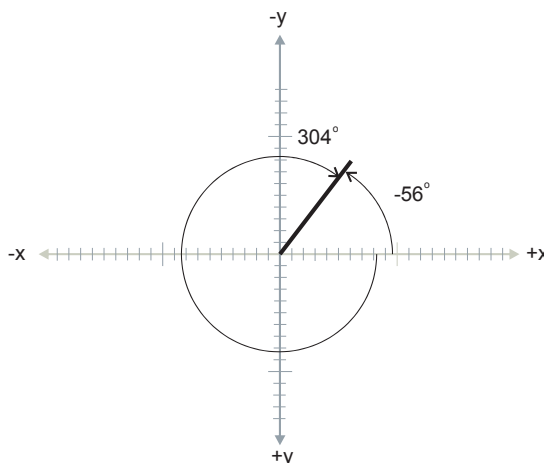
angle is a real number specifying the angle (in degrees) through which the line has been rotated. This is measured in a clockwise direction starting from the positive x-axis.

The real value returned by the function gives the x-coordinate of one end of the rotated line (the other end being at the origin).

The angle through which the line has been rotated may also be measured in a counter-clockwise direction, but is then specified as a negative value. This means that the expressions `Cos (304)` and `Cos (-56)` both return the same value (see FIG-9.28).

FIG-9.28

Clockwise and Counter-Clockwise Angles



Sin()

To determine the y-coordinate of our one unit line, we use the `sin ()` function which has the format shown in FIG-9.29.

FIG-9.29

`Sin()`

real `Sin` `(` `angle` `)`

where:

angle is a real number specifying the angle (in degrees) through which the line has been rotated. This is measured in a clockwise direction starting from the positive x-axis.

The real value returned by the function gives the y-coordinate of one end of the rotated line (the other end being at the origin).

Activity 9.17

Using *Calculator*, write down the y-coordinates of the four lines shown in Activity 9.16.

Dealing with Longer Lines

It's all very well to calculate the end point of a line which is one unit in length, but what about lines that are 2, 4 or 7.5 units long?

Actually, the calculation required is quite simple: if the line is twice as long, the coordinates of its end point are twice the value of those for a one unit line. If the line is four times longer, then the coordinate values are four times as large.

All of this can be simplified to:

$$\begin{aligned}\text{x-coordinate} &= \text{length of line} * \cos(\theta) \\ \text{y-coordinate} &= \text{length of line} * \sin(\theta)\end{aligned}$$

Activity 9.18

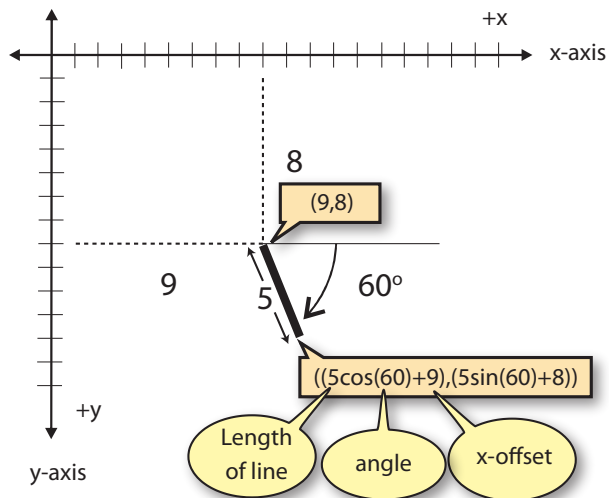
If a line is drawn from the origin and is 3.7 units in length, what are the coordinates of its end point after it has been rotated to an angle of 191.5° ?

Offset Lines

If a line whose fixed end is not positioned at the origin is rotated, calculating the coordinates of the moving end is done by calculating the x and y coordinates as before but then adding the x and y offset values to the results (see FIG-9.30).

FIG-9.30

Offset Lines



Activity 9.19

Calculate the actual coordinates of the rotating end of the line shown in FIG-9.29.

Using Cos() and Sin()

School may teach you the sine and cosine functions for no obvious practical reason, but when it comes to games programming, these are important operations. Using the `Sin()` and `Cos()` functions allows us to perform many operations on the screen graphics. For example, to rotate a sprite about a point on the screen. The program code in FIG-9.31 demonstrates how this is done by rotating a spot-shaped image

about the centre of the screen.

FIG-9.31

Rotating a Sprite

```
rem *** Load image ***
LoadImage(1,"Spot.png")
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Size sprite ***
SetSpriteSize(1,5,-1)
rem *** Position sprite offset from screen centre ***
SetSpritePosition(1,70,50)
angle = 0
do
    angle = (angle+1) mod 360
    SetSpritePosition(1,20*cos(angle)+50,20*sin(angle)+50)
    Sync()
loop
```

Activity 9.20

Start a new project called *Rotation*.

Compile the default code in order to create the *media* subfolder.

From the files you download with this book, copy the file *Spot.png* from the *AGKDownloads/Chapter9* folder into this project's *media* folder.

Change *main.agc* to match the code shown in FIG-9.31.

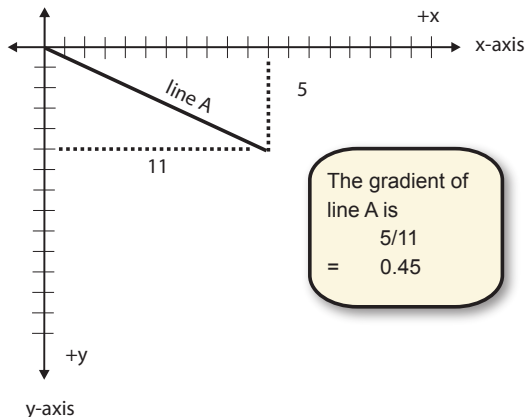
Test and save your project.

Tan()

The last of the traditional trigonometric functions is **tangent** or **tan**. Tangent measures the gradient (or steepness) of a line. Gradient of a line with one end fixed at the origin is just the y-coordinate of the other end of the line divided by the x-coordinate (see FIG-9.32).

FIG-9.32

Gradients



So a line parallel to the x-axis has a gradient of zero, a line at 45° to the x-axis has a gradient of 1 and a line at 90° has an infinite gradient. The `Tan()` function takes the angle of the line and returns its gradient.

FIG-9.33

Tan()

AGK's *Tan()* function has the format shown in FIG-9.33.

real Tan (angle)

where:

angle is a real number specifying the angle (in degrees) through which the line has been rotated. This is measured in a clockwise direction starting from the positive x-axis.

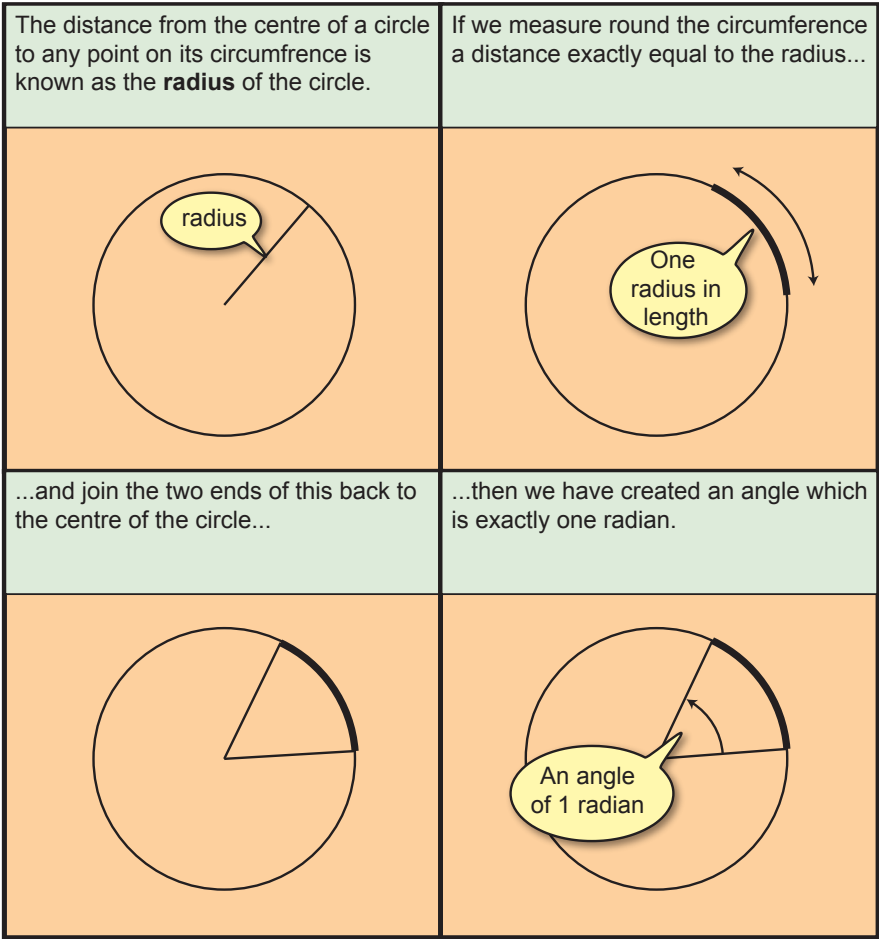
Degrees and Radians

No one knows with certainty why a full circular rotation is divided into 360 units known as **degrees** (or **degrees of arc**, to give them their full title). One theory is that ancient Persian civilizations used a calendar of 360 days (indicating a full rotation of the Earth about the Sun).

An alternative unit of measurement is the **radian**. FIG-9.34 explains how this measurement is derived.

FIG-9.34

Radians



AGK BASIC has a set of functions equivalent to *Cos()*, *Sin()* and *Tan()* called *CosRad()*, *SinRad()* and *TanRad()* which take an angle given in radians rather than degrees. The syntax for all three of these functions is shown in FIG-9.35.

FIG-9.35

CosRad()
SinRad()
TanRad()

real **CosRad** (**angle**)
real **SinRad** (**angle**)
real **TanRad** (**angle**)

where:

angle is a real number specifying the angle (in radians) through which the line has been rotated.

ACos(), ASin() and ATan()

If you already know the *x* or *y* coordinates of the end point of a line or the gradient of a line, but want to know the angle, then we can make use of the **ACos()**, **ASin()** or **ATan()** functions respectively. For example, looking back at FIG-9.31 we can see the gradient of the line is 5/11 but we don't know the angle the line makes to the *x*-axis. We can find out using the line

The mathematical names for these functions **arccosine**, **arcsine** and **arctangent**.

angle = ATan(5/11)

In Activity 9.16 we discovered the coordinates of point A (which was rotated by 50°) to be (0.643,0.766). Using the *x*-coordinate only, the line

angle = ACos(0.643)

will give a result of (approximately) 50. And using the *y*-coordinate only

angle ASin(0.766)

will also give the same result.

The syntax of the three statements are given in FIG-9.36.

FIG-9.36

ACos()
ASin()
ATan()

real **ACos** (**value**)
real **ASin** (**value**)
real **ATan** (**value**)

where:

value is a real number.

The value returned represents an angle given in degrees.

ACosRad(), ASinRad(), ATanRad()

If you need the angles to be returned in radians rather than degrees, you can make use of the **ACosRad()**, **ASinRad()** and **ATanRad()** functions which are shown in FIG-9.37.

FIG-9.37

ACosRad()
ASinRad()
ATanRad()

real **ACosRad** (**value**)
real **ASinRad** (**value**)
real **ATanRad** (**value**)

where:

value is a real number.

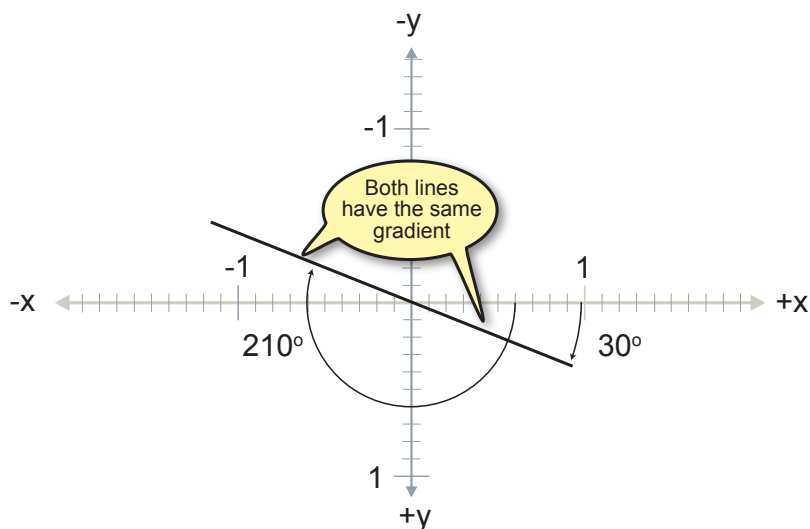
The value returned represents an angle given in radians.

ATanFull() and ATanFullRad()

The main problem with **ATan()** is that it cannot guarantee that the angle returned is the the correct one. Every gradient can be reproduced at exactly two angles. For example, a line at 30° and one at 210° have the same gradient (See FIG-9.38).

FIG-9.38

Angles and Gradients



The only way to differentiate between the two lines is to specify the end points of the line rather than the gradient value. This is the option offered by the **ATanFull()** and **ATanFullRad()** functions. The first returns the angle of the line in degrees, the second, in radians. The functions have the format shown in FIG-9.39.

FIG-9.39

ATanFull()
ATanFullRad()

real **ATanFull** ((x-coord , y-coord))

real **ATanFullRad** ((x-coord , y-coord))

where:

xcoord is a real number giving the x-coordinate of the line whose angle is to be found.

ycoord is a real number giving the y-coordinate of the line whose angle is to be found.

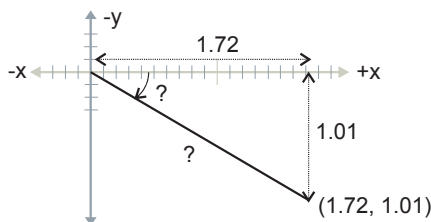
The angle returned is the angle that the specified line (whose second end is assumed to be at the origin), makes with the line from the origin to (0,1) - that is a line along the negative y-axis. Note that this is exactly 90° more than how all other angles are determined.

Sqrt()

If we started by knowing the end points of a line, could we work out the length of that line? Well, if we take a second look at what's going on when we calculate the value of a sine or cosine (see FIG-9.40), we can see how this calculation can be done.

FIG-9.40

End Points and Line Length



From the diagram, we can see that the end point coordinates actually represent the lengths of two sides of a right-angled triangle, so the length of the third side (the line we've drawn and the hypotenuse of the triangle) is given as:

$$\text{length of line} = \sqrt{\text{xcoord}^2 + \text{ycoord}^2}$$

Calculating the square of a value can be done with a line such as

```
xcoord * xcoord
```

or

```
xcoord^2
```

To calculate the square root, we might use

```
length = (xcoord^2 + ycoord^2) ^ 0.5
```

However, AGK BASIC provides a `Sqrt()` function which performs the same operation as `^0.5`. The statement's format is shown in FIG-9.41.

FIG-9.41

Sqrt()

real `Sqrt` (`(` value `)`)

where:

value is the real value whose square root is to be found. This cannot be a negative value.

So, the length of our line could be calculated as

```
length = Sqrt(xcoord^2 + ycoord^2)
```

Abs()

There are occasions when we want the value of a number without worrying about whether this is a positive or negative number.

In Chapter 4 we displayed the difference between a randomly generated number in the range 1 to 6 and the player's guess at what that number might be. This difference was calculated as:

```
diff = dice - guess
```

However, sometimes that difference would be a negative value when the guess was larger than the dice value. By using the **Abs()** function, which always returns the positive form of the argument, this problem can be eliminated:

```
diff = Abs(dice-guess)
```

FIG-9.42

Abs()

The format for the **Abs()** statement is given in FIG-9.42.

real **Abs** ((value))

where:

value is a real value.

The absolute value of *value* will be returned by the statement.

Ceil()

Returns the next integer value greater than or equal to the argument. Hence,

```
Ceil(12.1)
```

returns 13 while

```
Ceil(15.0)
```

returns 15.

But remember, when using a negative argument as in

```
Ceil(-14.9)
```

the function returns -14 (which is greater than -14.9) and not -15 (which is less than -14.9).

FIG-9.43

Ceil()

The **Ceil()** function has the format shown in FIG-9.43.

integer **Ceil** ((value))

where:

value is the real number whose value is raised to determine the return value.

Floor()

Complementing the **Ceil()** function is the **Floor()** function which returns the largest integer smaller than or equal to the function's parameter. This means that

```
Floor(12.1)
```

returns 12

```
Floor(15.0)
```

returns 15, and

```
Floor(-14.9)
```

returns -15.

FIG-9.44

Floor()

The `Floor()` function has the format shown in FIG-9.44.

integer `Floor` ((`value`))

where:

value is the real number whose value is lowered to determine the return value.

Trunc()

`Trunc()` is perhaps the simplest of the numeric functions since it returns the integral part of the real number argument, eliminating the fraction. So

```
Trunc(12.9)
```

returns 12 and

```
Trunc(-15.1)
```

returns -15.

FIG-9.45

Trunc()

The format for this statement is given in FIG-9.45.

integer `Trunc` ((`value`))

where:

value is the real number whose value is to be truncated.

Round()

Whereas the `Trunc()` function returns an integer by deleting the fraction part of the parameter, `Round()` returns an integer by rounding the parameter to the nearest integer. Hence,

```
Round(15.1)
```

returns 15 and

```
Round(15.6)
```

returns 16.

Rounding up happens for fractions of over 0.5 for positive value (.0 to 0.5 rounds down).

Where the absolute value of a negative number's fraction is over 0.5, the value is rounded down:

```
Round(-64.6) returns -65
```

FIG-9.46

Round()

The format for **Round()** is given in FIG-9.46.

integer **Round** ((value))

where:

value is a real number whose value is to be rounded to the nearest integer.

Fmod()

Whereas the **mod** operator returns the integer remainder when two integer values are divided, the **Fmod()** function returns the complete remainder (integral and fraction) when two real numbers are divided. Hence,

Fmod(7.0, 5.0)

returns 2.0 since 5.0 divides into 7.0 once with a remainder of 2.

Fmod(16.9, 5.1)

returns 1.6 since 5.1 divides into 16.9 3 times with a remainder of 1.6.

FIG-9.47

Fmod()

The syntax for the **Fmod()** function is shown in FIG-9.47.

real **Fmod** ((num , dem))

where:

num is a real value giving the numerator of the operation.

dem is a real value giving the denominator.

Activity 9.21

Give the value returned by each of the following function calls:

- | | | |
|-----------------------------|-------------------------|-------------------------|
| a) Sqrt (64) | b) Abs (-9) | c) Ceil (-9.1) |
| d) Floor (14.0) | e) Trunc (12.95) | f) Round (-16.9) |
| g) Fmod (-12.6, 3.2) | | |

Summary

- Cartesian coordinates use a horizontal x-axis and vertical y-axis to measure positions in a 2D space.
- These axes meet in the middle of that space at a point called the origin.
- All distances in the x and y directions are measured from the origin.
- By convention, points to the right of the origin on the x-axis are assigned positive values; points to the left, negative values.
- Points above the origin on the y-axis are assigned positive values; points below the origin, negative values.
- The axes divide 2D space into four quadrants.

- Any point in 2D space can be uniquely defined by specifying its position perpendicular to the x and y axes.
- A point's position in 2D space is known as the coordinates of the point and are given in the form

(distance along the x-axis, distance along the y-axis)

normally this description is shortened to

(x,y)

- On a computer, the positive section of the y-axis points down.
- A computer screen represents only part of quadrant 1 in 2D space.
- The `Cos()` function returns the cosine of a specific angle given in degrees.
- The `Sin()` function returns the sine of a specific angle given in degrees.
- The `Cos()` and `Sin()` values for the same angle give the end coordinates of a line one unit in length whose other end is at the origin.
- For a line a units in length coming from the origin and at an angle of θ° to the x-axis, the end coordinates are $(a*\cos(\theta), a*\sin(\theta))$.
- For a line of length a whose start point is at position (m,n) and lies at θ° to the x-axis, the other end's coordinates are given as $(a*\cos(\theta)+m, a*\sin(\theta)+n)$.
- The `Tan()` function returns the tangent of a specified angle given in degrees.
- Radians are an alternative way of measuring angles.
- One radian is the angle created when two radii of a circle are drawn in such a way that the distance along the arc of the circle's circumference from one radius to the other is exactly equal to the radius.
- The `CosRad()` function returns the cosine of a specific angle given in radians.
- The `SinRad()` function returns the sine of a specific angle given in radians.
- The `TanRad()` function returns the tangent of a specified angle given in radians.
- The `Acos()` function returns the angle of a line drawn from the origin with a specified end x-coordinate. The angle is given in degrees in the range 0° to 180° .
- The `Asin()` function returns the angle of a line drawn from the origin with a specified end y-coordinate. The angle is given in degrees in the range -90° to $+90^\circ$.
- The `Atan()` function returns the angle of a line with a specified gradient. The angle will lie in the range -90° to $+90^\circ$.
- The `AcosRad()` function returns the angle of a line drawn from the origin with a specified end x-coordinate. The angle is given in radians (0 to 2π).
- The `AsinRad()` function returns the angle of a line drawn from the origin with a specified end y-coordinate. The angle is given in radians ($-\pi$ to π).
- The `AtanRad()` function returns the angle of a line with a specified gradient. The angle is given in radians ($-\pi$ to π).

- The `ATanFull()` function returns the angle of a line (with specified end points) to the negative part of the y-axis. The result is in degrees (0° to 360°).
- The `ATanFullRad()` function returns the angle of a line (with specified end points) to the negative part of the y-axis. The result is in radians (0 to 2π).
- The `Sqrt()` function returns the square root of the function argument.
- The `Abs()` function returns the absolute value of the function argument.
- The `Trunc()` function returns an integer value calculated as the truncated value of the parameter.
- The `Round()` function returns an integer value calculated as the rounded value of the parameter.
- The `Fmod()` function returns the remainder produced by the division of two real values.

Solutions

Activity 9.1

The program should display the random string that was generated and its length.

Activity 9.2

b\$ would be set to 1-BY-1.

Activity 9.3

The original version of *Letters*:

```
#include "StringLibrary.agc"

text$ = RandomString(-1)
Print("Original string is: "+text$)
for c = 1 to Len(text$)
    Print(Mid(text$,c,1))
next c
Sync()
do
loop
```

This will begin by displaying the original string, then each letter of that string on separate lines.

To display the letters in reverse order, the `for` loop needs to decrement from `Len(text$)` down to 1. So the new code is:

```
#include "StringLibrary.agc"

text$ = RandomString(-1)
for c = Len(text$) to 1 step -1
    Print(Mid(text$,c,1))
next c
Sync()
do
loop
```

The final version counts the number of E's in the string:

```
#include "StringLibrary.agc"

text$ = RandomString(-1)
count = 0
for c = Len(text$) to 1 step -1
    if Mid(text$,c,1) = "E"
        inc count
    endif
next c
Print("Original string is: "+text$)
PrintC("It contains ")
PrintC(count)
PrintC(" E's")
Sync()
do
loop
```

Activity 9.4

The code for *ASCIITable*:

```
for c = 32 to 126
    rem *** Display number ***
    PrintC(c)
    PrintC(" is the ASCII code for ")
    rem *** Display character ***
    Print(Chr(c))
    rem *** If 25th update screen and wait 5 secs
    ⚡***
    if (c-31) mod 25 = 0
        Sync()
        Sleep(5000)
    endif
next c
Sync()
do
loop
```

Activity 9.5

Code for *CountZero*:

```
rem *** Generate random number ***
num = Random(1000,65000)
rem *** Convert to a string ***
num$= str(num)
rem *** Start count at zero ***
count = 0
rem *** Check each character ***
for c = 1 to Len(num$)
    rem *** If it's a 0, increment count ***
    if Mid(num$,c,1) = "0"
        inc count
    endif
next c
rem *** Display details ***
Print("Original number "+num$)
Print("Contains "+str(count)+" zeros")
Sync()
do
loop
```

Activity 9.6

Create a new project called *Conversions*.

Compile the default code.

Copy the file *Buttons.png* and *Buttons subimages.txt* to the *media* folder (you'll find a copy in *TestButtons*).

Copy *Buttons.agc* into the *Conversions* folder.

In the *setup.agc* file, change height to 1024 and width to 768.

Code *main.agc* as:

```
#include "Buttons.agc"

SetUpButtons()
rem *** Get value from buttons ***
num = GetButtonEntry()
rem *** Display number in dec, binary and hex ***
Print("Number (base 10) : "+Str(num))
Print("Number (base 2) : "+Bin(num))
Print("Number (base 16) : "+Hex(num))
Sync()
do
loop
```

Activity 9.7

No solution required.

Activity 9.8

Create a new project called *FunctionTester*.

Copy *StringLibrary.agc* into the *FunctionTester* folder.

The code for *FunctionTester* is:

```
rem *** Test Pos Function ***
#include "StringLibrary.agc"
text$ = RandomString(30)
post = Pos(text$,"D")
Print("String is "+text$)
Print("D at position "+Str(post))
Sync()
do
loop

rem *** Find Position of character in string ***
function Pos(s$, f$)
    rem *** result stays at 0 if no match found ***
    result = 0
    rem *** Make sure we're looking for a single
    ⚡character ***
    first$ = Mid(f$,0,1)
    rem *** FOR each character in s$ DO ***
    for c = 1 to Len(s$)
        rem *** IF that character matches what we're
        ⚡after THEN ***
        if Mid(s$,c,1) = first$
            rem *** Set result to this position and exit
            ⚡loop ***
```

```

        result = c
    exit
endif
next c
endfunction result

```

Changing the character searched for to a lowercase letter will guarantee that the letter is not found. The program will display zero for the position found. A better option would be to check for zero being returned with code such as:

```

#include "StringLibrary.agc"
text$ = RandomString(30)
post = Pos(text$, "D")
if post <> 0
    Print("String is "+text$)
    Print("D at position "+Str(post))
else
    Print("D not found in text")
endif
Sync()
do
loop

```

Activity 9.9

In the Projects Panel, right-click the project name and select **Add files** from the pop-up menu.

Select *StringLibrary.agc* from the listed files. That file will then be listed in the *Sources* part of the project.

Double-click on the *StringLibrary.agc* file in the Projects Panel. This will open a tab for the file's source code in the edit area.

Copy the code for function *Pos()* from *main.agc* and paste it into *StringLibrary.agc* after the existing function.

Select **Files|Save everything**

Activity 9.10

Updated code for *FunctionTester*:

```

rem *** Test Pos Function ***
#include "StringLibrary.agc"

text$ = RandomString(30)
count = Occurs(text$, "S")
Print("String is "+text$)
Print("S occurs "+Str(count)+" times")
Sync()
do
loop

rem *** Return how often f$ occurs in s$ ***
function Occurs(s$, f$)
    rem *** None found so far ***
    result = 0
    rem *** Make sure only one character ***
    first$ = Mid(f$, 0, 1)
    rem *** FOR each character in s$ Do ***
    for c = 1 to Len(s$)
        rem *** if it matches req'd character, add 1 to
        ↳ result ***
        if Mid(s$, c, 1) = first$
            result = result + 1
        endif
    next c
endfunction result

rem *** Find Position of character in string ***
function Pos(s$, f$)
    rem *** result stays at 0 if no match found ***
    result = 0
    rem *** Make sure we're looking for a single
    character ***
    first$ = Mid(f$, 1, 1)
    rem *** FOR each character in s$ DO ***
    for c = 1 to Len(s$)
        rem *** IF that character matches what we're
        after THEN ***
        if Mid(s$, c, 1) = first$

```

```

        rem *** Set result to this position and exit
    loop ***
        result = c
        exit
    endif
next c
endfunction result

rem *** Return how often f$ occurs in s$ ***
function Occurs(s$, f$)
    rem *** None found so far ***
    result = 0
    rem *** Make sure only one character ***
    first$ = Mid(f$, 0, 1)
    rem *** FOR each character in s$ Do ***
    for c = 1 to Len(s$)
        rem *** if it matches req'd character, add 1 to
        ↳ result ***
        if Mid(s$, c, 1) = first$
            result = result + 1
        endif
    next c
endfunction result

```

The code for *Occurs()* should be copied from *main.agc* and pasted into *StringLibrary.agc*.

Activity 9.11

Code for *FunctionTester* (previous functions are not shown):

```

text$ = "ABCDEFGHGI"
text$ = Insert(text$, "XX", 2)
Print("String is "+text$)
Sync()
do
loop

rem *** Inserts f$ at position p in s$ ***
function Insert(s$, f$, p)
    rem *** If invalid position, result is original
    ↳ string ***
    if p < 1 or p > Len(s$)+1
        result$ = s$
    else
        rem *** split s$ into two parts & insert f$
        ↳ in between ***
        result$ = Left(s$, p-1)
        result$ = result$ + f$
        result$ = result$ + Right(s$, Len(s$) - (p-1))
    endif
endfunction result$

```

Changing the line

```
text$ = Insert(text$, "XX", 2)
```

to

```
text$ = Insert(text$, "XX", 12)
```

will return the original text since the insert position given is invalid.

Copy and paste the code for the routine into *StringLibrary.agc*.

Activity 9.12

Code for *FunctionTester* (previous functions are not shown):

```

text$ = "ABCDEFGHGI"
text$ = Delete(text$, 3, 5)
Print("String is "+text$)
Sync()
do
loop

rem *** Delete num characters from s$ starting at
position st ***
function Delete(s$, st, num)
    rem *** if invalid position, result is original
    ↳ string ***
    if st < 1 or st > Len(s$)
        result$ = s$
    else

```



```

rem *** Set result to the part of s$ to the
↳left of ***
rem *** the section to be deleted ***
result$ = Left(s$, st-1)
rem *** IF not deleting to the end of s$, ***
rem *** add right section ***
if st+num-1 <= Len(s$)
    result$ = result$+Right(s$,Len(s$)-
↳(st+num-1))
endif
endif
endfunction result$

```

Changing

```
text$ = Delete(text$,3,5)
```

to

```
text$ = Delete(text$,13,5)
```

will return the original string since the delete position is invalid.

When the number of characters to be deleted is greater than the number available, all characters after the start position are deleted. Hence,

```
text$ = Delete(text$,3,15)
```

returns

```
AB
```

Copy the functions code and paste it into *StringLibrary.agc*.

Activity 9.13

FUNCTION NAME	:	Replace
PARAMETERS		
In	:	s : string
	:	sr : character
	:	p : integer
Out	:	result : string
PRE-CONDITION	:	1 <= p <= Len(s)
DESCRIPTION	:	result is equal to s except that the p th character of result is sr.

Code for *FunctionTester* (previous functions are not shown):

```

text$ = "ABCDEFGH"
text$ = Replace(text$,"X",3)
Print("String is "+text$)
Sync()
do
loop

rem *** Replace the pth character in s$ with rs$ ***
function Replace(s$, rs$, p)
rem *** If invalid position ***
rem *** return original string ***
if p < 1 or p > Len(s$)
    exitfunction s$
endif
rem *** If rs$ more than one character use left-
↳most character ***
rs$ = Left(rs$,1)
rem *** Calculate result ***
result$ = Left(s$,p-1)+rs$+Right(s$,Len(s$)-p)
endfunction result$

```

A line such as

```
text$ = Replace(text$,"X",13)
```

will return the original string since the position specified is invalid.

Copy and paste the code into *StringLibrary.agc*.

Activity 9.14

At 0° the x-coord is 1

At 90° the x-coord is 0

Activity 9.15

cos(0)	=	x-coord	=	1
cos(90)	=	x-coord	=	0
cos(30)	=	x-coord	=	0.866
cos(70)	=	x-coord	=	0.342

Activity 9.16

cos(50)	=	x-coord	=	0.643
cos(168)	=	x-coord	=	-0.978
cos(213)	=	x-coord	=	-0.839
cos(304)	=	x-coord	=	0.559

Activity 9.17

sin(50)	=	y-coord	=	0.766
sin(168)	=	y-coord	=	0.208
sin(213)	=	y-coord	=	-0.545
sin(304)	=	y-coord	=	-0.829

Activity 9.18

x coord = $3.7\cos(191.5)$ = -3.626
y coord = $3.7\sin(191.5)$ = -0.738

Activity 9.19

x coord = $5\cos(60)+9$ = 11.5
y coord = $5\sin(60)+8$ = 12.330

Activity 9.20

No solution required.

Activity 9.21

a) 8	b) 9	c) -9
d) 14	e) 12	f) -17
g) -3.0		

In this Chapter:

- ☐ The Limitations of Standard Variables
- ☐ The Concept of Arrays
- ☐ Declaring Arrays
- ☐ Initialising Arrays
- ☐ Accessing Array Elements
- ☐ Array Subscripting
- ☐ Arrays and Counting
- ☐ Arrays and Non-Repeating Values
- ☐ Arrays and Shuffling
- ☐ Arrays and Sorting
- ☐ Arrays and Searching
- ☐ Multi-dimensional Arrays
- ☐ Arrays as Function Parameters

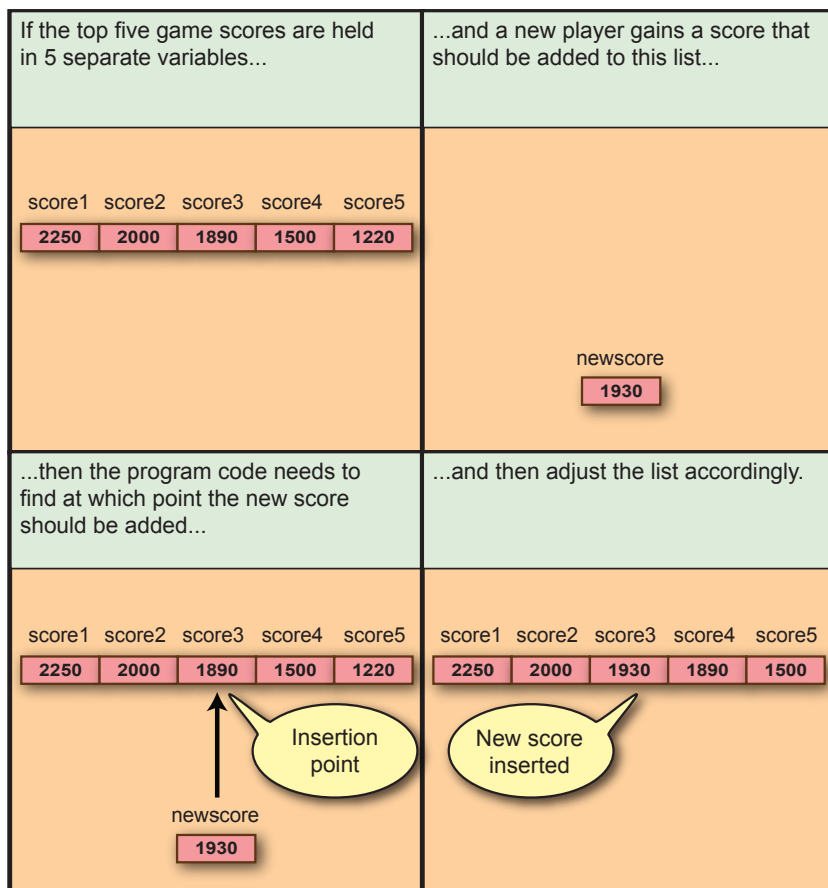
Problems with Simple Variables

There are certain tasks which are very difficult or long-winded when we try to do them using the normal variables we've been dealing with up to now. For example, it's common for a video game to retain the top five scores but, from what we know at the moment, we'd have to set up one variable for each score to be saved.

When a player finishes a game, the program then has to decide if the player's score should be recorded in the top five and, if so, at what position. If the new score is good enough to be recorded as a highest score, then the list must be updated. The whole process is shown in FIG-10.1.

FIG-10.1

Using Regular Variables



Notice that what had been the third and fourth highest scores, have now moved down one position and that the score of 1220 has been lost from the top five.

We need to develop an algorithm which can perform the above task for all possible values which might be placed within the top five scores. One possible structured English solution could use the lines:

```
IF
  newscore > score1:
    score5 = score4
    score4 = score3
```

```

        score3 = score2
        score2 = score1
        score1 = newscore
    newscore > score2:
        score5 = score4
        score4 = score3
        score3 = score2
        score2 = newscore
    newscore > score3:
        score5 = score4
        score4 = score3
        score3 = newscore
    newscore > score4:
        score5 = score4
        score4 = newscore
    newscore > score5:
        score5 = newscore
ENDIF

```

Activity 10.1

Assuming the following values

```

score1 = 2250 score2 = 2000 score3 = 1890 score4 = 1500
score5 = 1220

newscore = 1900

```

work your way through the algorithm given above to check that the expected result is obtained.

The algorithm is a bit long-winded, but just about acceptable. Now imagine that we had the top ten scores to retain. What would the algorithm look like then? It's going to be long - very long. Luckily, there is a better way to achieve what we're after - **arrays**.

One Dimensional Arrays

Array Concepts

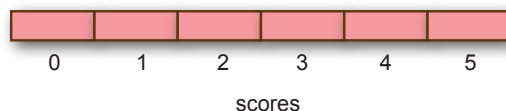
An **array** is a named data variable capable of storing several values at the same time. It is a collection of **elements** or **cells**. Each of these elements holds a single value - just like the regular variables we have used in our previous programs.

Each cell within an array is numbered. The first cell is numbered cell zero, the next cell 1, etc. Exactly how many cells an array contains is determined when the array is first set up.

FIG-10.2 shows how we might visualise a 6 element array called *scores*.

FIG-10.2

Visualising Arrays



The individual cells within the array are identified by a combination of the array name and the cell's number which is known as the **subscript**. The subscript is enclosed within square brackets. For example, the second cell within the array shown

above is identified using the term

```
scores[1]
```

Declaring an Array

Whereas we are free to introduce a standard variable at any point in a program, we need to tell the compiler in advance that we intend to use an array. This is known as an **array declaration**. An array declaration begins with the keyword `dim` followed by the name we wish to assign to the array. The only additional piece of information required is the subscript for the final element of the array - this determines how many elements the array is to contain. So, to set up the *scores* array as shown in FIG-10.2, we would use the declaration

```
dim scores[5]
```

This creates a 6 cell array with the cells numbered 0 to 5.

Arrays can be declared to hold values of any of the types we can use for regular variables: integer, real or string. For example,

```
dim averages#[10] rem *** 11 element real array ***  
dim names$(19)   rem *** 20 element string array ***
```

Every cell within the array can then hold a single value of the specified type. It is not possible to create an array with cells of differing types.

Initialising Arrays

When an array is first set up, every cell in the array contains the value zero (or an empty string when using string arrays). But it is possible to specify a different starting value for each cell when declaring the array by extending the array declaration. For example, the line

```
dim numbers[3] = [12, 0, -6, 8]
```

will create the array setup shown in FIG-10.3.

FIG-10.3

Array Initialisation

12	0	-6	8
0	1	2	3
numbers			

If there are too many values specified within the braces, the surplus values are ignored; if there are too few values supplied, then the cells which have not specifically been assigned a value are set to zero.

Accessing Array Elements

We cannot perform operations on an array as if it were a single entity. For example, it would be invalid to try to display all the values held in an array with a statement such as

```
Print(numbers)
```

Instead, we must deal with the individual elements within the array. So, to display the value in the first element in the array *numbers*, we would write

```
Print(numbers[0])
```

To assign a value to the next element we could use a statement such as

```
numbers[1] = 4
```

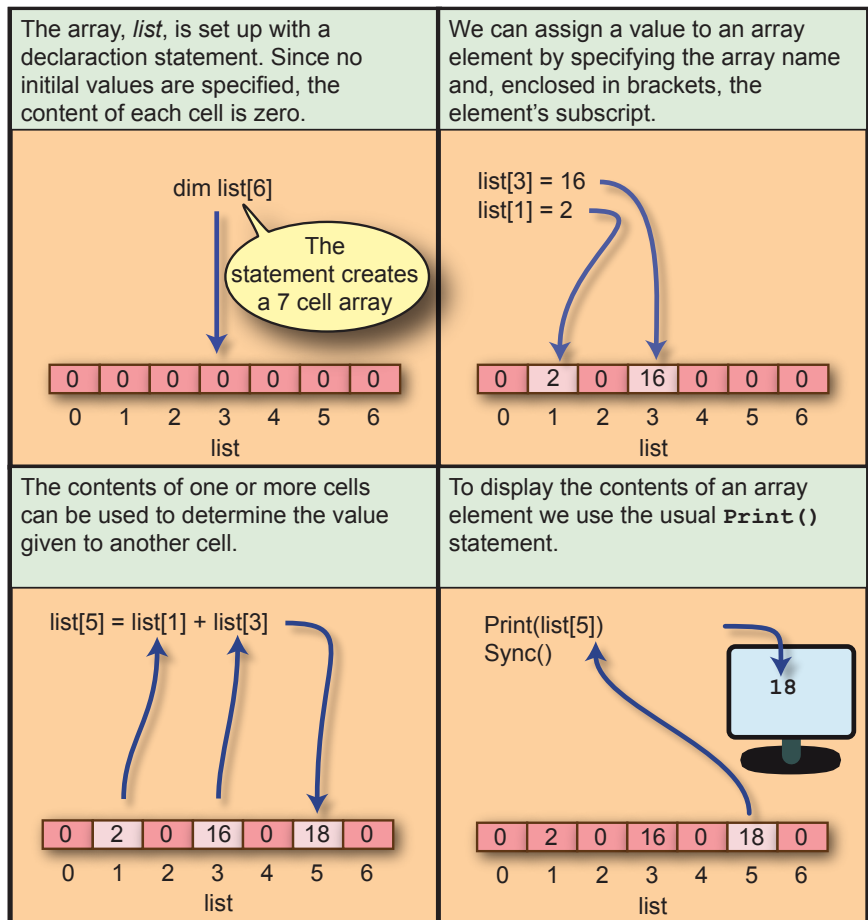
and we could check if the last element contained a value of less than zero with the line

```
if (numbers[3] < 0)
```

In fact, we can use an array element in any statement where we might use a simple variable of the same type. Some more examples are shown in FIG-10.4.

FIG-10.4

Accessing Array
Elements



You must ensure that the subscript you supply is a valid one; the compiler will not check that the subscript value is within a range compatible with the size of the array created. Hence, code such as

```
dim list[6]
list[7]= 21; rem *** Subscript too high ***
```

will be compiled but when the program is running it will halt when the assignment statement is reached and display a message of the form

Subscript out of bounds at line 2

What Makes Arrays Powerful

If what we've seen up to now was all that could be achieved by arrays, they would be of little more use than simple variables. For example, if we were to simulate the throwing of four dice using four integer variables, we could use the lines:

```
dice1 = Random(1,6)
dice2 = Random(1,6)
dice3 = Random(1,6)
dice4 = Random(1,6)
```

Using an array would require the lines:

```
dim dice[4]
dice[1] = Random(1,6)
dice[2] = Random(1,6)
dice[3] = Random(1,6)
dice[4] = Random(1,6)
```

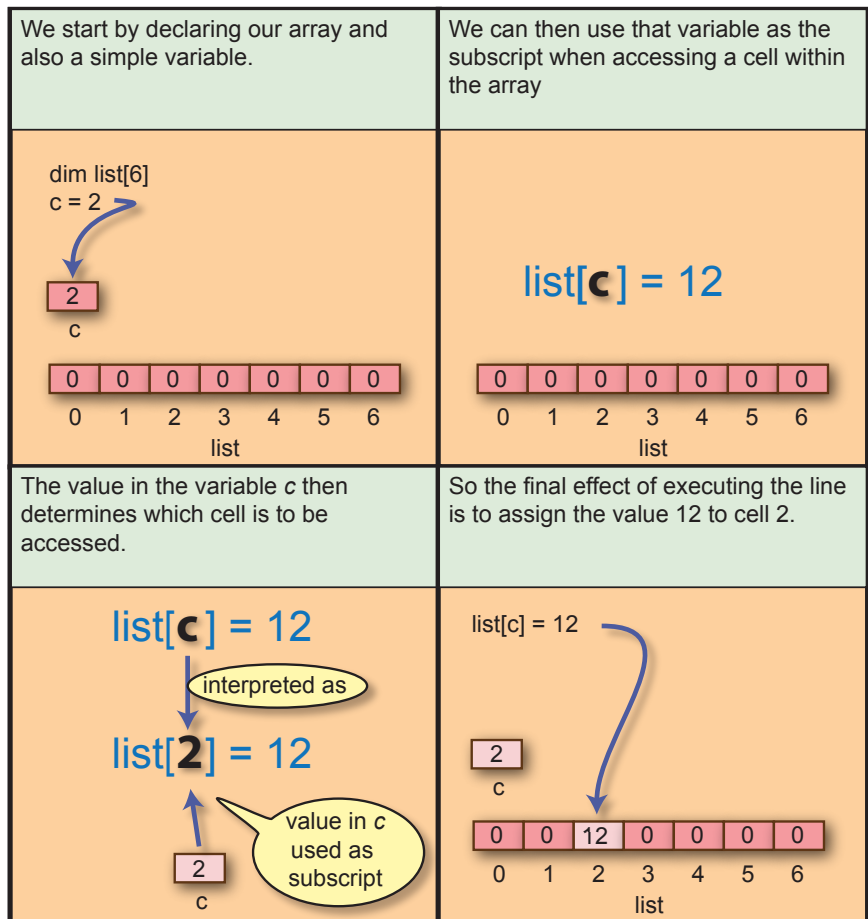
➡ `dice[0]` is unused.

Both segments are equally long winded.

What adds power to the array is the fact that the subscript need not be given as a fixed value. Instead, we are free to use a variable. The value of that variable then determines the value of the subscript and hence which element within the array is to be accessed (see FIG-10.5).

FIG-10.5

Variable Subscripts



If the contents of the variable *c* are changed, then it follows that the array element being accessed will also change.

We need to take only one further step to realise how we can use arrays to create shorter code for our earlier problem of reading in four values.

With the code

```
for c = 0 to 6
```

the variable *c* will, as the loop repeats itself, take on first the value 0, then 1, 2, etc. and finally 6.

So, returning to our dice code, if we use a variable in conjunction with array access, we can assign our four values using the code

```
dim dice[4]
for c = 1 to 4
    dice[c] = Random(1,6)
next c
```

As *c* changes value each time the loop iterates, so the term `dice[c]` in the assignment statement will reference a different element of the array *dice*.

Activity 10.2

Start a new project called *Arrays01*, and, using the code given above, create a complete program which stores the values obtained by four dice throws in the array *dice* (ignore element zero).

By adding a second `for` loop to your code, get the program to display the contents of the array. Test and save your project.

Array Element Zero

Every array always has an element zero - as we have already seen. But there are times when the clarity of an algorithm is better served by ignoring this element. For example, we used elements *dice[1]* to *dice[4]* to store our dice throws, ignoring *dice[0]*. If we want to store information based on the months of the year, we would probably set up the appropriate array

```
dim months[12]
```

and use *months[1]* to *months[12]* since this corresponds to the months of the year.

Of course, there is no reason why we could not use elements *dice[0]* to *dice[3]* and *months[0]* to *months[11]* for our data, but doing that detracts slightly from how we might normally think. And that means we are more likely to make mistakes in our program logic and hence, in these cases, using element zero is probably best avoided.

The only downside of ignoring element zero is that we end up making our arrays one element larger than they need to be. This seems like a small price to pay given the memory available on modern devices.

Array Subscript Options

We've already seen that an array subscript can be given in the form of a constant as

in `dice[1]` or as a variable (`dice[c]`), but it can also be given in the form of an arithmetic expression. For example, in the code

```
dim values[20]
p = 3
values[p*2] = 42
```

will store the value 42 in `values[6]` - which is the seventh cell within the array.

We can even use the contents of one cell as the subscript. So, in the code

```
dim values[20]
values[0] = 9
values[values[0]] = 4
```

will result in the value 4 being stored in `values[9]`.

Activity 10.3

State the contents of each cell in the array *numbers* after the following code has been executed.

```
dim numbers[8]
for p = 0 to 8
    numbers[p] = p*2
next p
numbers[numbers[2]-1] = 23
```

Using Arrays

We've already seen a simple example of how we might make use of an array, but arrays can be used in many more ways. Some examples of how arrays can be used to help create an efficient solution to a problem are shown in this section.

Problem: Multiple Counts

One of the tests used to make sure that a dice is not bias is to check that, for a large number of throws, each number should appear approximately the same number of times.

Solution:

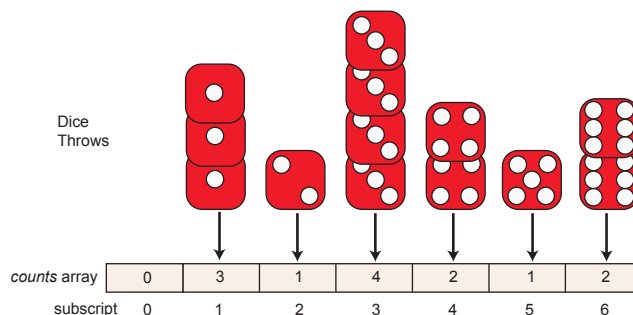
We can use an array to keep count of how often each number occurs. Cell 1 will hold a count of how often the number 1 is thrown, cell 2 the number of times 2 is thrown, etc (see FIG-10.6).

A dice is said to be bias if each number does not have the same likelihood of being thrown.

FIG-10.6

Counts Concept

Since the diagram shows only a small number of throws, the distribution of each number can vary more widely.



The structured English for our solution could be written as

```
FOR 1000 times DO
  Throw dice
  Add 1 to appropriate count
ENDFOR
Display all 6 counts
```

The code for the program is given in FIG-10.7.

FIG-10.7

Keeping Multiple
Counts

```
dim counts[6]
rem *** Throw the dice 1000 times ***
for c = 1 to 1000
  rem *** Throw dice ***
  dicethrow = Random(1,6)
  rem *** Add to appropriate count ***
  inc counts[dicethrow]
next c

rem *** Display each count ***
for c = 1 to 6
  Print(Str(c)+" occurred "+Str(counts[c])+" times")
next c
Sync()
do
loop
```

Activity 10.4

Start a new project *DiceCount*.

Modify *main.agc* to match the code given in FIG-10.7. Test and save your project.

Some games make use of a 10 sided dice. Modify your program so that it will generate numbers in the range 1 to 10 and count how often each value occurs.

As you see, modifying the code for a 10-sided dice requires changes in several lines. To avoid this we could set the number of sides as a named constant.

Modify your code to use a named constant called *SIDES* for the number of sides on the dice.

Now change the code to deal with a 20 sided dice. How many lines of code need to be modified to handle this?

Problem: Generating Random Non-Repeating Values

Many countries run lottery systems. The simplest of these require you to choose 6 unique numbers in the range 1 to 49. Of course, we can easily get the computer to generate and display six numbers in this range, but we also need to make sure that none of the six numbers are the same.

Solution:

To ensure that there are no duplicate values from the second number onwards, we

need to check that the generated number has not already been selected. One way to do this is to set up an array containing a cell for each number that might be generated. Initially all the cells contain zero, but when a number is selected the corresponding cell's value is set to 1. When a number is generated, it can only be added to the list of selected values if its corresponding cell contains a zero at that point (see FIG-10.8).

FIG-10.8

Unrepeated Random
Values: Concept

The array contains an element for each number that can be generated (1 to 49). Initially, every cell contains zero.

numbers array

0	0	0	0	0	0			0	0	0	0	0	0	
0	1	2	3	4	5			44	45	46	47	48	49	

When a value is generated, the corresponding cell is set to 1.

0	0	0	0	0	0			0	1	0	0	0	0	
0	1	2	3	4	5			44	45	46	47	48	49	

If 45 is generated, *numbers*[45] is set to 1.

Other generated values are only accepted if the corresponding cell in *numbers* contains a zero.

0	0	0	0	0	0			0	1	0	0	0	0	
0	1	2	3	4	5			44	45	46	47	48	49	

If 4 is generated, it is accepted because *numbers*[4] contains zero.

Once accepted, the matching cell is set to 1.

0	0	0	0	1	0			0	1	0	0	0	0	
0	1	2	3	4	5			44	45	46	47	48	49	

numbers[4] set to 1.

The structured English for our solution would be:

```

Set all cells to 0
Generate a random number in the range 1 to 49
Set the corresponding cell to 1
Display the value
FOR 5 times DO
    REPEAT
        Generate a random number
    UNTIL the corresponding cell is zero
    Set the corresponding cell to 1
    Display the number
ENDFOR

```

The program for this is shown in FIG-10.9.

FIG-10.9

Unrepeated Random
Values: Code

```
#constant HIGHEST = 49

dim lottery[HIGHEST]

rem ***Generate number ***
number = Random(1,HIGHEST)

rem *** Set corresponding cell ***
lottery[number] = 1

rem *** Display value ***
Print(number)
rem *** FOR 5 times DO ***
for c = 1 to 5
    rem *** Generate an unselected number ***
    repeat
        rem ***Generate number ***
        number = Random(1,HIGHEST)
    until lottery[number] = 0
    rem *** Set corresponding cell ***
    lottery[number] = 1
    rem *** Display value ***
    Print(number)
next c
Sync()
do
loop
```

Activity 10.5

Start a new project, *Lottery*. Modify *main.agc* to match the code in FIG-10.9 and test the program.

There is really no need to treat the first number any differently from the remaining five. Modify the code so that all 6 numbers are generated within the **for** loop.

The code displays the numbers as they are generated rather than in ascending order. Modify the code so that the six numbers are displayed in ascending order.

(HINT: You will need to remove the existing **Print()** statements from the code.)

Problem: Shuffling

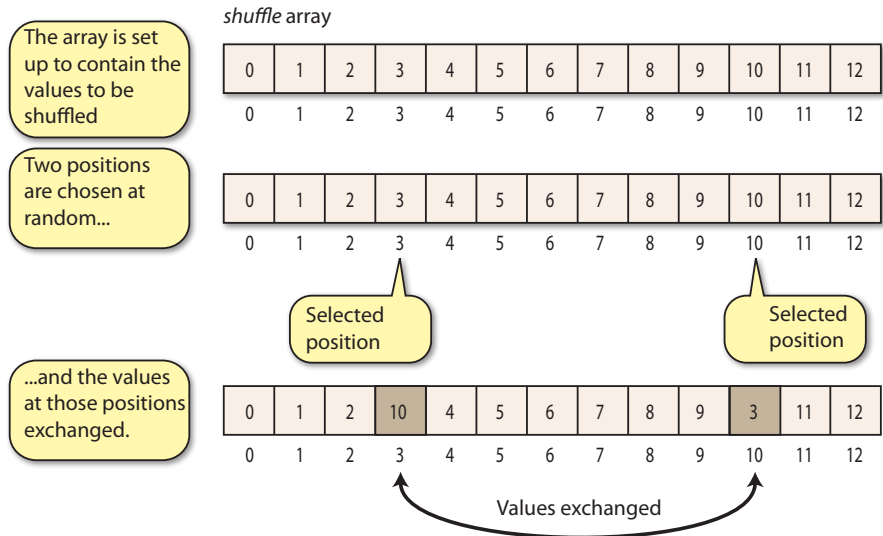
Many applications require items to be re-arranged in a random order. For example, your MP3 player probably offers a shuffle option which will play music tracks in a random order and shuffling is mandatory for almost every card game.

Solution:

If we start by storing a set of values in an array (these may represent music track numbers or playing card values), then we can create a shuffle effect by taking the values at two randomly selected positions within the array and swapping them over (see FIG-10.10).

FIG-10.10

Shuffling: Concept



If we continue to do this many more times, the items will have been effectively shuffled.

The structured English is:

```

Set up all values within an array
FOR 200 times DO
    Generate a first random subscript
    Generate a second random subscript
    Swap the values held at the subscript positions
ENDFOR

```

The code for shuffling an array of 20 values is given in FIG-10.11.

FIG-10.11

Shuffling: Code

```

dim list[20]

rem *** Set up values in array ***
for c = 1 to 20
    list[c] = c
next c
rem *** Shuffle ***
for c = 1 to 200
    rem *** Generate two subscript values ***
    sub1 = Random(1,20)
    sub2 = Random(1,20)
    rem *** Swap values at these positions ***
    temp = list[sub1]
    list[sub1] = list[sub2]
    list[sub2] = temp
next c
rem *** Display shuffled items ***
for c = 1 to 20
    PrintC(Str(list[c])+ " ")
next c
Sync()
do
loop

```

Activity 10.6

Start a new project, *Shuffle*, and implement the code in FIG-10.11. After testing, modify the program so that the contents of *list* are displayed before and after the shuffle.

To simulate a card pack, we would need a 52 element list. The numbers 0 to 12 could represent ace to king of hearts; 13 to 25 diamonds; 26 to 38 spades; and 39 to 51 clubs. Modify the program to shuffle a deck of cards and display the first six “cards” in the list.

It would be better if we could display the value and suit of a card rather than just a number. For example, displaying *2 of diamonds* rather than *14*. Modify your program to do this. For the moment, Ace, Jack, Queen and King can be displayed as 1, 11, 12, and 13 of the appropriate suit. (HINT: Use the division and modulo operators (`/` and `mod`) to determine the suit and value of a card.

A final improvement would be to display the names Ace, Jack, Queen and King as appropriate. Test and save your project.

Problem: Handling an Array that is not Full

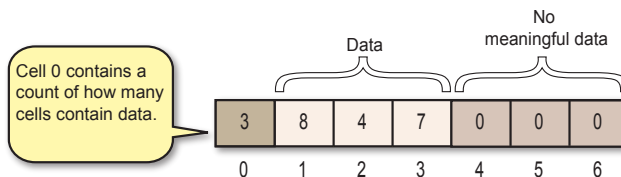
There are times when we will set up an array with space enough to hold a specific number of values, but initially not all the cells will contain meaningful data. For example, a game which remembers the top 5 scores, will contain no top scores when first played. In these situations, we may want to access only the elements of the array in which data has already been placed and so we need to know which cells contain data.

Solution:

One way to handle this problem is to use element zero in the array to keep a count of how many cells in the array contain data (see FIG-10.12).

FIG-10.12

A Data Count



The main steps involved in this setup are shown in FIG-10.13.

FIG-10.13

Using a Data Count

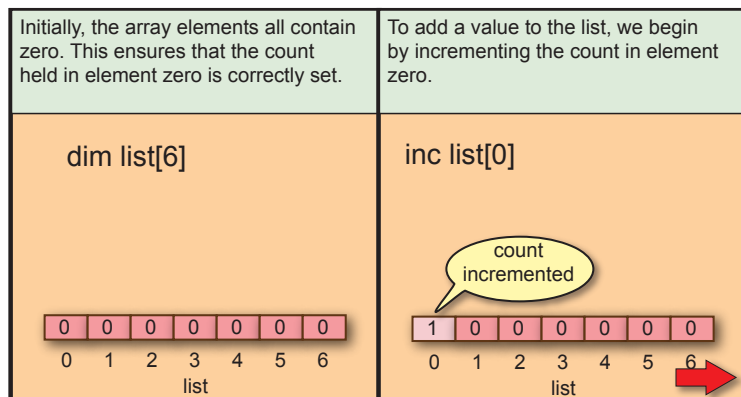
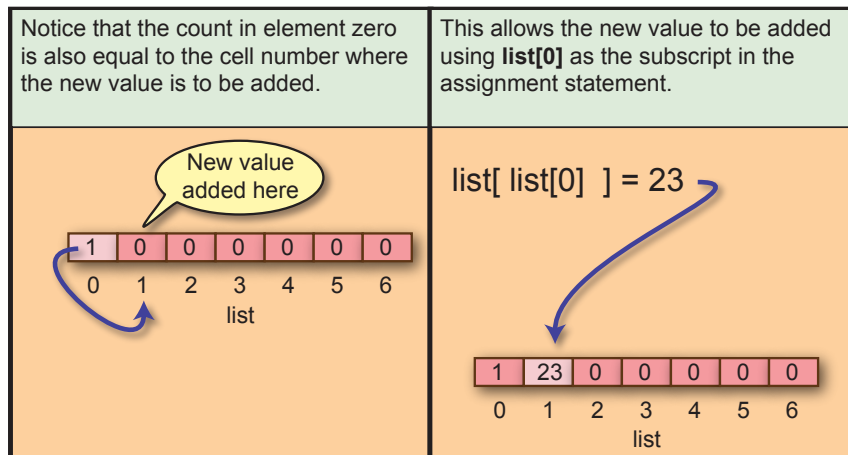


FIG-10.13

(continued)

Using a Data Count



The only check that is required when adding a new value is that the array must not already be full.

The logic required to insert a value into the list can be written in structured English as:

```

Get value to be added
IF the list is not full THEN
    Add 1 to the count in element zero
    Insert the new value at the end of the existing data
ELSE
    Display message "List is full"
ENDIF
  
```

A menu-driven program demonstrating how values are added to an array using the technique described above is given in FIG-10.14. Note that the program allows four options: add a value, display how many values are held, display the values held, and quit.

FIG-10.14

Implementing a Data Count

```

#include "Buttons.agc"

#constant SIZE 5
dim list[SIZE]

/** Repeat until quit selected **
SetupButtons()
repeat
    /** Display menu **
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - QUIT")
    /** Get option **
    Print("Enter option required(1-4) ")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 4)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
  
```



FIG-10.14

(continued)

Implementing a Data
Count

```

    /*** Execute option ***/
select option
  case 1: /*** Add a new value to the list ***/
    Print("Enter value to be added : ")
    Sync()
    Sleep(2000)
    value = GetButtonEntry()
    if list[0] < SIZE
      inc list[0]
      list[list[0]] = value
    else
      Print("List is full")
    endif
    Sync()
    Sleep(2000)
  endcase
  case 2: /*** Display the number of items in the list ***/
    Print("The list contains "+Str(list[0])+" entries")
    Sync()
    Sleep(2000)
  endcase
  case 3: /*** Display the contents of the list ***/
    if (list[0] = 0)
      Print("The list is empty")
    else
      Print("Values held are")
      for c = 1 to list[0]
        PrintC(Str(list[c])+" ")
      next c
    endif
    Sync()
    Sleep(2000)
  endcase
  case 4: /*** Quit program ***/
    Print("Quitting program in 2 seconds")
    Sync()
  endcase
endselect
until option = 4
Sleep(2000)
end

```

Activity 10.7

Start a new project called *DataCount* and implement the code given in FIG-10.14.

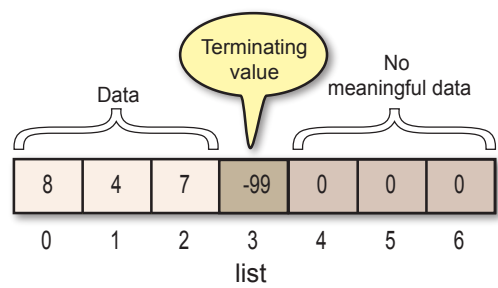
Remember to copy the three files needed to use the *Buttons* functions.

Test and save your project.

Keeping a count of the number of entries in an array is only one way of handling the problem of keeping tabs on just how many elements within an array contain meaningful data. A second approach is to use a “marker” value in the cell following the last value held in the array. For example, we might follow the actual data by a value of, say, -99 (see FIG-10.15).

FIG-10.15

A Sentinel Value

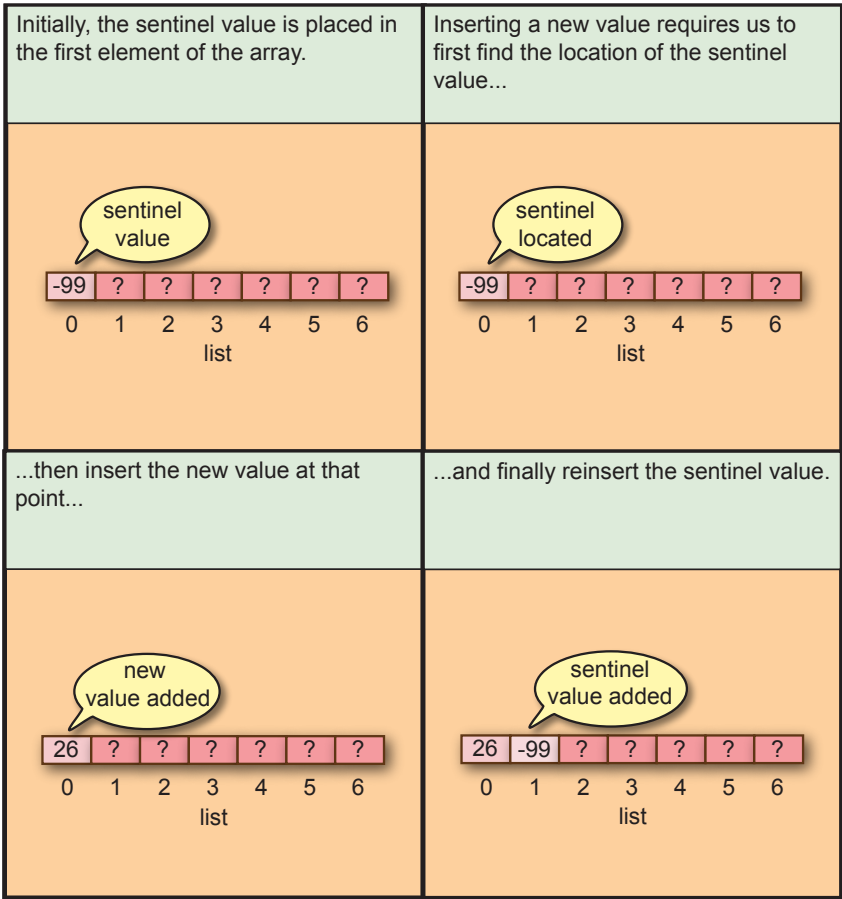


The marker value, often known as the **sentinel**, must be chosen with care. Obviously, we cannot use a value which can occur within the actual data, since such an occurrence would be assumed to be the terminating value. For example, we could safely choose the value -1 as a terminating value if we were sure that all the actual data values were positive.

The main characteristics of this approach are shown in FIG-10.16.

FIG-10.16

Using a Sentinel Value



The complex part of this operation is locating the sentinel value within the array. After several values have been added, there is no easy way of knowing the sentinel's position. To find its location, we must search through the contents of the array.

Searching a list of values for some specific entry is one of the commonest requirements in a software application. There are many ways of searching a list. For the moment,

we will content ourselves by examining only one of these.

If we are looking for the value -99 in a list of values, we can compare -99 with each value in the list and stop when we find a match. This can be achieved by the code:

```
post = 0
while list[post] <> -99
    inc post
endwhile
```

Once the insert position has been found, we need to insert the new value and place -99 to its new position. Assuming we are using the version of the `while` loop given above, this would be achieved using the lines

```
list[post] = value
list[post+1] = -99;
```

We can determine if the list is empty using the expression

```
if list[0] = -99
```

and if it is full using

```
list[SIZE] = -99
```

To count the number of entries in the list, we are forced to search for the sentinel. Its position in the list will be equal to the number of entries. For example, initially -99 is in cell 0 and there are zero entries in the list; when -99 is stored in cell 3 there will be three entries in the list (occupying cells 0, 1 and 2).

Activity 10.8

Using the program you created in Activity 10.7 as a guide, create a new project called *SentinelData* which makes use of a sentinel-based list.

The program should retain the same four options: allowing a value to be added to the data, displaying the number of values already stored, displaying the actual contents of the array, and a quit option.

Test and save your project.

Problem: Inserting a Value into an Array

In the previous problem, new values were inserted at the end of the existing data. However, there are many circumstances when the new value will be required to be positioned elsewhere within that data. As we will see, inserting a new value within existing data causes new problems.

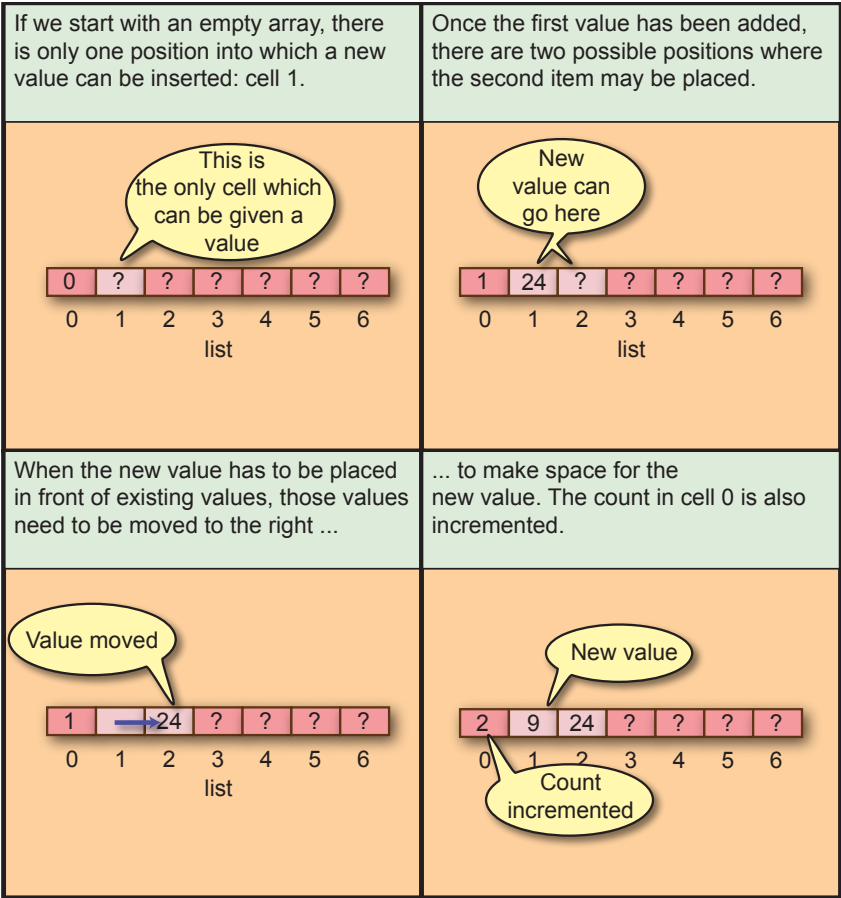
Solution:

When we want to add a value between existing values (just as we might add a character into a misspelled word when using a word processor) then we need to create space at the insertion point by moving other values out of the way.

FIG-10.17 shows the steps involved. for a count-based array.

FIG-10.17

Adding a Value to an Array



The value 24 is actually copied into `list[2]`, so `list[1]` is not empty (as suggested by the diagram), but still contains the value 24. This will be overwritten when the new value is inserted.

Activity 10.9

Assuming an array is in the state shown in the diagram below



in which cells may a new value be positioned?

Assuming the value 77 is to be placed in cell 3, show, with the aid of diagrams, the state of the array after:

- a) Existing values have been moved to make space for the new value.
- b) The new value has been inserted.

When a new item is being added we need to acquire not only its value, but also the cell number into which it is to be placed. This can be done using the code:

```

Print("Enter new value")
Sync()
Sleep(1000)
value = GetButtonEntry()
Print("Enter its position")
Sync()
Sleep(1000)
post = GetButtonEntry()
while (post < 1 or post > list[0]+1)
    Print("Invalid position. Must be in the range 1 to ")
    Print(Str(list[post]+1))
    Print("Re-enter position")
    Sync()
    Sleep(1500)
    post = GetButtonEntry()
endwhile

```

Notice that the code includes a check to insure that the insert position is valid.

To free space for the new value, we need to move all those values between *post* and *list[0]* up one position within the array. This is done using the following code:

```

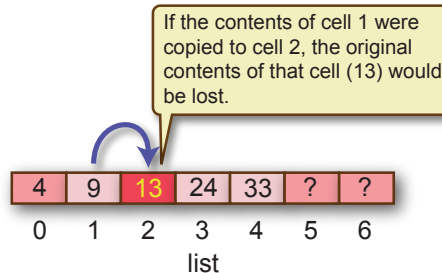
for current = list[0] to post step -1
    new = current + 1
    list[new] = list[current]
next current

```

As you can see, it is necessary to move the value at the end of the list first, otherwise you would overwrite the next value (see FIG-10.18).

FIG-10.18

Moving Data



All that remains now is to add the new value and increment the count held in *list[0]*:

```

list[post] = value
inc list[0]

```

Activity 10.10

Modify your *DataCount* project so that, when new data is entered, the program requests an insert position for the data and places the new data at the specified position.

Test and save your project.

Problem: Recording the Top Scores

We started this chapter by looking at what was involved in maintaining a record of the top five scores in a video game. We have now learnt enough about arrays and the techniques employed when using them to tackle that earlier problem. But there are a

couple of new problems to handle:

- When a top score is achieved, the point at which it is inserted in a list is determined by the existing values in that list.
- Once five scores have been recorded within the list, any new score that is added will mean that the lowest score will be eliminated from the list.

Solution:

To insert a new top score, we need to search down the list to find the first entry with a value which is less than the one we wish to add.

```
post = 1
while list[post] >= newscore
    inc post
endwhile
```

This will give us the insert position as long as the *newscore* is greater than at least one of the existing top scores. However, there would be a problem if this was not the case : the **while** loop would fail to terminate! We could try adding a second condition to the loop so that it terminates if we arrive at the end of the array:

```
post = 1
while list[post] >= newscore and post <= SIZE
    inc post
```

Unfortunately, this leaves us with another problem: when *post* is incremented for the last time, it will be set to 6, then, as we loop back and test the first condition

```
list[post] >= newscore
```

we will be trying to access *scores[6]* - a cell which does not exist.

One way to solve this problem is to make the array one cell larger than we need:

```
dim list[SIZE++1]
```

which would mean that the array does contain a cell identified as *scores[6]* although we will never make use of that cell (we only need the top 5 scores).

A second option is to re-organise the conditions within the **while** statement:

```
while post <= SIZE and list[post] >= newscore
```

AGK BASIC implements **short-circuit evaluation**. When two conditions are ANDed together and the first is evaluated to false, the second condition is not tested. This means we won't attempt to access a non-existent element of *list*.

Now we have a situation exactly as before with a value and a position at which it is to be inserted. The only other change we need to make is to allow the lowest value in *scores[5]* to be eliminated when the lower values are shifted to make space for the new value. This can be done by making the first shift from cell 4 to cell 5 (the last cell) thereby overwriting the value previously held in cell 5.

The code for this is:

```
for current = 4 to post step -1
    new = current + 1
```

```

        list[new] = list[current]
    next current

```

A more flexible solution would be to initialise *current* to *SIZE* rather than 4. This would allow the number of high scores to be changed without having to alter any code other than the definition of *SIZE*. This gives us the new line:

```

    for current = SIZE to post step -1

```

When we first begin to store the highest scores, *scores* will not be full and so we must increment the count held in *list[0]*, but once we have 5 high scores, the count should remain fixed. This requirement can be handled by the lines

```

    if list[0] < SIZE
        inc list[0]
    endif

```

The complete code for inserting a new high score is:

```

rem *** Get new score ***
Print("Enter new score")
Sync()
Sleep(1000)
newscore = GetButtonEntry()
rem *** Find insertion point ***
post = 1
while(post <= SIZE and list[post] >= newscore)
    inc post
endwhile
rem *** Create space for new score ***
for current = SIZE to post step -1
    new = current + 1
    list[new] = list[current];
next current
rem *** Add new new score ***
list[post] = newscore;
rem *** Increment count ***
if list[0] < SIZE
    inc list[0]
endif

```

With a few modifications we can make use of the program FIG-10.14 to test our code.

Activity 10.11

Start a new project called, *TopScores*. Compile the default code and copy the files required by the *Button* functions into the appropriate folders in the new project. Copy all the code from the latest version of *DataCount* to *TopScore*'s *main.agc*.

In *TopScores*, modify case 1 in the **select** statement to match the code given above. (There is no requirement to check if the array is full.) Add an extra element to array *list*.

Test your program using the following data for the high scores:

23000, 11000, 17000, 46000, 9000

Display the list to make sure it is in descending order.

Add a new score 31000, and check that the score of 9000 is removed from the list.

Problem: Deleting a Value

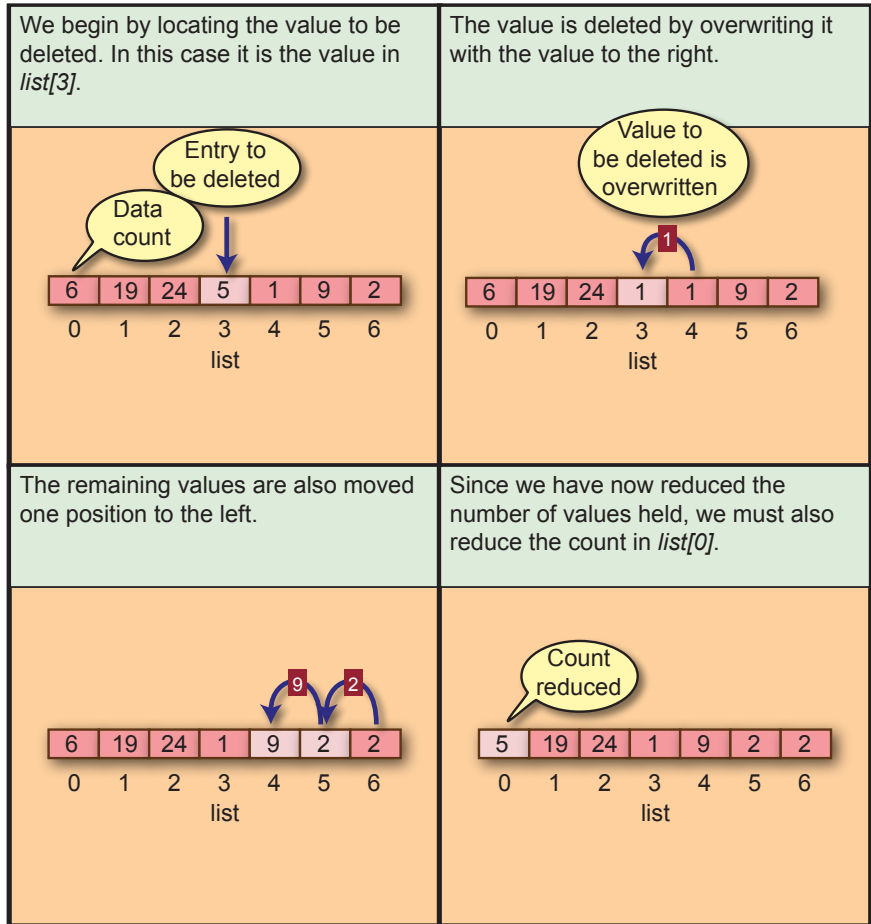
Some situations require an item of data to be deleted from an existing list. Sometimes we need to find and delete a specific value; other times way may want to delete the entry at a specific position in the list irrespective of its value.

Solution:

To delete a value from a list, we must first locate that value and then eliminate it from the list. FIG-10.19 shows the stages involved.

FIG-10.19

Deleting Data



Notice that *list*[6] retains a copy of the final value in the list, but this will have no effect on our code since, by reducing the count, the content of *list*[6] is no longer regarded as part of the valid data.

If we want to delete the value held at position *post*, then the logic required to move the other data items is:

```
rem *** Delete entry by moving subsequent entries to left ***
for current = post+1 to list[0]
    list[current-1] = list[current]
next current
rem *** Reduce count ***
dec list[0]
```


When we accept a value for *post*, we must ensure that we are attempting to delete from a position that contains data, so the following code is required:

```
rem *** Get position ***
Print("Enter position of item to be deleted")
Sync()
Sleep(1000)
post = GetButtonEntry()
while post < 1 or post > list[0]
    Print("The position is invalid. Re-enter.")
    Sync()
    Sleep(1000)
    post = GetButtonEntry()
endwhile
```

Activity 10.12

In *DataCount*, change the displayed menu so that the last two options are

- 4 - Delete from position
- 5 - QUIT

Make use of the code given above to add a new **case 4**: option in the **select** statement which deletes the data from a specified position in the list.

Change **case 5** : to be the quit option.

Change the condition in the **until** line to be **option = 5**.

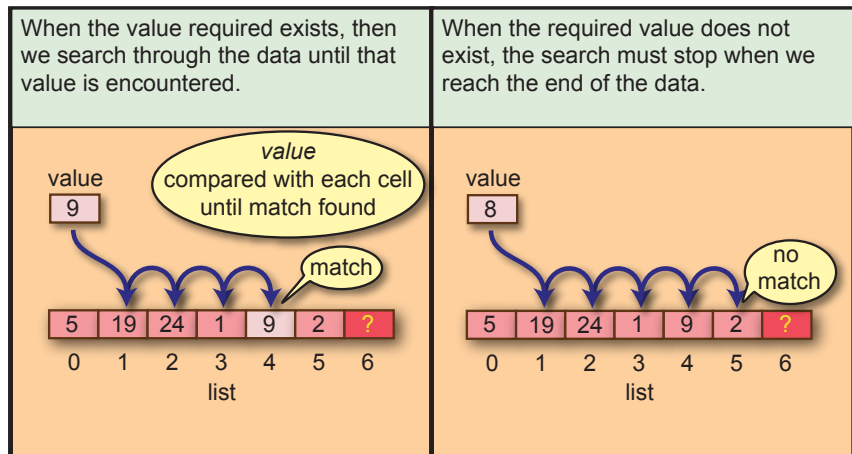
Test your new code by first setting up 5 values in the list and then deleting the last item of data, the third item of data and the first item of data. Display the contents of the list after each delete.

Save this updated version of *DataCount*.

When we want to delete a specific value from the list, we must first locate that value. This requires logic very like that required for locating the sentinel in a list. However, whereas we knew the sentinel would always appear in a sentinel-terminated list, we cannot make the same guarantees for user-selected values (see FIG-10.20).

FIG-10.20

Searching



Our new logic must allow for both possibilities.

What we require can be described in structured English as:

```
Get value to be deleted
Start at beginning of list
WHILE not arrived at end of data AND value to be deleted not found DO
    Move to next entry in the list
ENDWHILE
```

This translates into AGK BASIC as:

```
/** Enter value to be deleted **
Print("Enter value to be deleted")
Sync()
Sleep(1000)
value = GetButtonEntry()

/** Search for value in list **
post = 1
while (post <= list[0] and list[post] <> value)
    inc post
endwhile
```

Once the **while** loop has been completed, we need to check if the match was found. If it was, the cell we stopped at will contain a value matching the required value and this can be checked with the code:

```
if list[post] = value
```

Having found a match, the contents of the array can then be re-arranged to delete the specified entry and the count decremented.

Activity 10.13

Modify *Datacount* so that the value to be deleted - not its position - is entered. If the value to be deleted cannot be found, an appropriate message should be displayed.

Test Data: list 3, 6, 9, 12
 Values to delete : 6, 12, 2

Display the list content after each deletion.

Save your project.

Problem: Converting Numbers to Text

A common requirement in a program handling dates is to display a day or a month in text rather than as a number. For example, sometimes we want to display the word *September* rather than number 9 when showing a date.

Solution:

To perform this task we can set up an array containing the months of the year in text form with the text for each month in the appropriate cell; so, cell 1 would contain the word *January*, cell 2 *February*, etc.

In the code given below, we have a string array, local to a function, which contains

the names of the months of the year, When supplied with the month of the year as a parameter, the function returns the corresponding string.

```
Print(MonthOfYear(8))
Sync()
do
loop

function MonthOfYear(v)
  if v < 1 or v > 12
    exitfunction ""
  endif
  dim month$[12] =["","January","February","March","April",
    ↵ "May","June","July","August","September","October",
    ↵ "November","December"]
  result$ = month$[v]
endfunction result$
```

Activity 10.14

Start a new project called *UsingStringArrays*, and implement the code given above.

Test and save the project.

Activity 10.15

In project *Shuffle* we made use of two `select` statements to display the card suit and the card values.

Modify *Shuffle* so that it makes use of string arrays to perform these tasks.

Test and save the project.

Dynamic Arrays

Sometimes it is not possible to know how large an array should be at the time we are writing a program. For example, let's say we need an array to hold the score achieved by each player in a multi-player game.

The number of elements needed in the array depends on the number of people who are actually playing on any specific occasion. To handle this situation, AGK BASIC allows the size of an array to be set using a variable. A snippet of the code required is shown below:

```
rem *** Find out how many people are playing ***
Print("Enter the number of players")
Sync()
Sleep(1000)
noofplayers = GetButtonEntry()
rem *** Set up an array of that size ***
dim scores[noofplayers]
```

Activity 10.16

Start a new project called *DynamicArray*. The program should create an array of between 5 and 12 cells (this number to be chosen at random). Place a random value (between 1 and 20) in each cell and finally, display the contents of the array.

The undim Statement

If a program creates a particularly large array with thousands of elements, or has very many arrays, then it will occupy significant amounts of memory. This in turn may slow down the speed at which your program runs. To avoid this, it is possible to delete arrays which are no longer required using the `undim` statement which has the format shown in FIG-10.21.

FIG-10.21

The `undim` Statement

`undim` `arrayname` `[` `]`

where:

arrayname is the name of the array to be deleted. The array must have been created earlier using a `dim` statement.

For example, if, at the start of a program we had created an array with the line

```
dim list[20]
```

then we could destroy that array later in our code using the line

```
undim list[ ]
```

Multi-dimensional Arrays

Could we represent the game of chess using an array? The problem here is that the chess board has rows and columns, while the arrays we have encountered up to now are just one long list of values. Luckily AGK BASIC allows us to create arrays which have both rows and columns. These are called **two-dimensional arrays**.

To do this we need to start by declaring our array using an extended form of the `dim` statement in which the number of rows and columns are specified. For example, if we wanted to keep the 6 best scores for 5 different players, we could set up a 5 row by 6 columns array called *scores* using the line

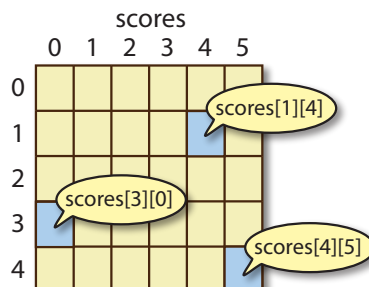
```
dim scores[4,5]
```

This would create the structure shown in FIG-10.22.

FIG-10.22

The *scores* 2D Array

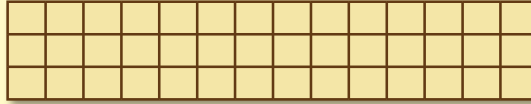
This time we will make use of the row zero and column zero in the array.



Activity 10.17

Write the declarations necessary for the array structures pictured below (assume all hold `integer` values; use any name you wish).

a)



b)



c)



To access an individual element within a two-dimensional array, we must specify the array name and the row and column numbers. The row and column values are separated by a comma. For example, we could store the value 23 in the top-left cell of the array *marks*, using the code:

```
marks[0,0] = 23
```

Unfortunately, there is no option to initialise multi-dimensional arrays.

We saw earlier how we could use a `for` loop to access each element of a one-dimensional array in turn. That same technique can be used to access a two dimensional array. The only difference this time is that we need to employ two `for` loops.

Returning to our *scores* array, we could store a random value in each cell using the following code:

```
for row = 0 to 4
  for col = 0 to 5
    scores[row,col] = Random(1,20)
  next col
next row
```

Activity 10.18

Start a new project called *Using2DArrays*. In *main.agc*, create the array *scores* with 5 rows and 6 columns.

Make use of the code given above to store a random value in the range 1 to 20 in each cell of the array.

Add more code to display the contents of each cell in the array. Display values from the same row on one line.

Test and save your program.

3-Dimensional Arrays and Higher

There are situations where we may need an array with even more dimensions. For example, if our players played with three levels of difficulty, then we would need an array which had three dimensions (5 players, 6 scores, 3 levels). We would define such an array with the statement:

```
dim scores[4,5,2]
```

AGK BASIC allows for arrays of up to 8 dimensions.

Arrays and Functions

Arrays cannot be used as function parameters nor as return values. If you want to make use of a non-local array within a function, then you must declare the array as a global variable as in the line

```
global dim numbers[20]
```

Summary

- Arrays can be used to hold a collection of values.
- Every value in an array must be of the same type.
- Arrays are created using the `dim` statement.
- The number of elements in an array can be specified as a constant, variable, or expression.
- Using a variable or expression to set the array's size allows that size to be varied each time the program is run.
- Numeric arrays are created with the value zero in every element.
- String arrays are created with empty strings in every element.
- The space allocated to an array can be freed using the `undim` statement.
- An array element is accessed by giving the array named followed by the element's subscript value enclosed in square brackets.
- The first element in an array has a subscript value of zero.
- The subscript can be a constant, variable or expression.
- Arrays can have up to eight dimensions.
- An array cannot be passed as a parameter to a function.
- A function cannot return an array as a result.

Solutions

Activity 10.1

The condition `newcore > score3` is true, so the lines executed will be

```
score5 = score4
score4 = score3
score3 = newscore
```

Activity 10.2

The code for *Array01*:

```
rem *** Using Arrays ***

rem *** Declare array ***
dim dice[4]
rem *** Store values in array ***
for c = 1 to 4
    dice[c] = Random(1,6)
next c
rem *** Display the values held ***
for c = 1 to 4
    Print(dice[c])
next c
Sync()
do
loop
```

Activity 10.3

The `for` loop will result in the following values being stored in *numbers*:

0,2,4,6,8,10,12,14,16

The final statement uses the contents of `numbers[2]` - which is 4 - minus 1 (which gives a result of 3) as the subscript in the expression

```
numbers[numbers[2]-1] = 23
```

so the line can be interpreted as

```
numbers[3] = 23
```

so the final contents of the array are

0,2,4,23,8,10,12,14,16

Activity 10.4

Modified code for *DiceCount*:

```
rem *** Dice throw counter ***
rem ** Declare array ***
dim counts[10]
rem *** Throw the dice 1000 times ***
for c = 1 to 1000
    rem *** Throw dice ***
    dicethrow = Random(1,10)
    rem *** Add to appropriate count ***
    inc counts[dicethrow]
next c
rem *** Display each count ***
for c = 1 to 10
    Print(Str(c)+" occurred "+Str(counts[c])+" times")
next c
Sync()
do
loop
```

DiceCount with a constant:

```
rem *** Dice throw counter ***
#constant SIDES 10
rem ** Declare array ***
dim counts[SIDES]
```

```
rem *** Throw the dice 1000 times ***
for c = 1 to 1000
    rem *** Throw dice ***
    dicethrow = Random(1,SIDES)
    rem *** Add to appropriate count ***
    inc counts[dicethrow]
next c

rem *** Display each count ***
for c = 1 to SIDES
    Print(Str(c)+" occurred "+Str(counts[c])+" times")
next c
Sync()
do
loop
```

The only change required to deal with a 20-sided dice is:

```
#constant SIDES 20
```

Activity 10.5

Modified code for *Lottery*:

```
#constant HIGHEST = 49

dim lottery[HIGHEST]

rem *** FOR 6 times DO ***
for c = 1 to 6
    rem *** Generate an unselected number ***
    repeat
        rem ***Generate number ***
        number = Random(1,HIGHEST)
        until lottery[number] = 0
    rem *** Set corresponding cell ***
    lottery[number] = 1
    rem *** Display value ***
    Print(number)
next c
Sync()
do
loop
```

Modified code for *Lottery* (numbers in ascending order):

```
#constant HIGHEST = 49

dim lottery[HIGHEST]

rem *** FOR 6 times DO ***
for c = 1 to 6
    rem *** Generate an unselected number ***
    repeat
        rem ***Generate number ***
        number = Random(1,HIGHEST)
        until lottery[number] = 0
    rem *** Set corresponding cell ***
    lottery[number] = 1
next c
rem *** Display subscript of cells containing 1 ***
for c = 1 to HIGHEST
    if lottery[c] = 1
        Print(c)
    endif
next c
Sync()
do
loop
```

Activity 10.6

Modified code for *Shuffle*:

```
dim list[20]

rem *** Set up values in array ***
for c = 1 to 20
    list[c] = c
next c
Print("Original Order")
rem *** Display contents ***
for c = 1 to 20
    PrintC(Str(list[c])+" ")
next c
Print("")
```

```

rem *** Shuffle ***
for c = 1 to 200
    rem *** Generate two subscript values ***
    sub1 = Random(1,20)
    sub2 = Random(1,20)
    rem *** Swap values at these positions ***
    temp = list[sub1]
    list[sub1] = list[sub2]
    list[sub2] = temp
next c
rem *** Display shuffled items ***
Print("Shuffled order")
for c = 1 to 20
    PrintC(Str(list[c])+" ")
next c
Sync()
do
loop

```

Card version of *Shuffle*:

```

#constant SIZE 52
dim list[SIZE]

rem *** Set up values in array ***
for c = 1 to SIZE
    list[c] = c-1
next c
Print("")
rem *** Shuffle ***
for c = 1 to SIZE *20
    rem *** Generate two subscript values ***
    sub1 = Random(1,SIZE)
    sub2 = Random(1,SIZE)
    rem *** Swap values at these positions ***
    temp = list[sub1]
    list[sub1] = list[sub2]
    list[sub2] = temp
next c
rem *** Display shuffled items ***
Print("First six cards")
for c = 1 to 6
    PrintC(Str(list[c])+" ")
next c
Sync()
do
loop

```

Notice that the value stored in *list[c]* is *c-1* (so that we are storing 0 to 51 rather than 1 to 52).

Named suits version of *Shuffle*:

```

#constant SIZE 52
dim list[SIZE]
rem *** Set up values in array ***
for c = 1 to SIZE
    list[c] = c-1
next c
Print("")
rem *** Shuffle ***
for c = 1 to SIZE *20
    rem *** Generate two subscript values ***
    sub1 = Random(1,SIZE)
    sub2 = Random(1,SIZE)
    rem *** Swap values at these positions ***
    temp = list[sub1]
    list[sub1] = list[sub2]
    list[sub2] = temp
next c
rem *** Display shuffled items ***
Print("First six cards")
for c = 1 to 6
    PrintC(Str(list[c] mod 13+1) + " of ")
    select list[c] / 13
        case 0:
            Print("Hearts")
        endcase
        case 1:
            Print("Diamonds")
        endcase
        case 2:
            Print("Spades")
        endcase
        case 3:
            Print("Clubs")
        endcase
    endselect

```

```

next c
Sync()
do
loop

```

When the value of the card is displayed in the statement

```
PrintC(Str(list[c] mod 13 + 1) + " of ")
```

the expression *list[c] mod 13* makes sure we have a value in the range 0 to 12. Since this is one less than the actual value of the card, we add 1 to the value (with the term *+1*).

The expression *list[c] / 13* in the *select* statement determines the suit. Hearts cards have values between 0 and 12, so any of these values will give an answer of 0 when divided by 12 (remember integer division is performed); 13 to 25 is the diamonds (division by 12 gives a result of 1); etc. So the *select*'s expression will give a result between 1 and 4 giving the suit of the card.

The named cards version of *Shuffle*:

```

#constant SIZE 52
dim list[SIZE]

rem *** Set up values in array ***
for c = 1 to SIZE
    list[c] = c-1
next c
Print("")
rem *** Shuffle ***
for c = 1 to SIZE *20
    rem *** Generate two subscript values ***
    sub1 = Random(1,SIZE)
    sub2 = Random(1,SIZE)
    rem *** Swap values at these positions ***
    temp = list[sub1]
    list[sub1] = list[sub2]
    list[sub2] = temp
next c
rem *** Display shuffled items ***
Print("First six cards")
for c = 1 to 6
    select list[c] mod 13+1
        case 1:
            PrintC("Ace")
        endcase
        case 11:
            PrintC("Jack")
        endcase
        case 12:
            PrintC("Queen")
        endcase
        case 13:
            PrintC("King")
        endcase
        case default
            PrintC(list[c] mod 13+1)
        endcase
    endselect
    PrintC(" of ")
    select list[c] / 13
        case 0:
            Print("Hearts")
        endcase
        case 1:
            Print("Diamonds")
        endcase
        case 2:
            Print("Spades")
        endcase
        case 3:
            Print("Clubs")
        endcase
    endselect
next c
Sync()
do
loop

```

The new *select* statement displays the appropriate term for cards with values 1, 11, 12, or 13, all other cards have their numeric value displayed.

Activity 10.7

No solution required.

```

endselect
until option = 4
Sleep(2000)
end

```

Activity 10.8

Code for SentinelData:

```

#include "Buttons.agc"

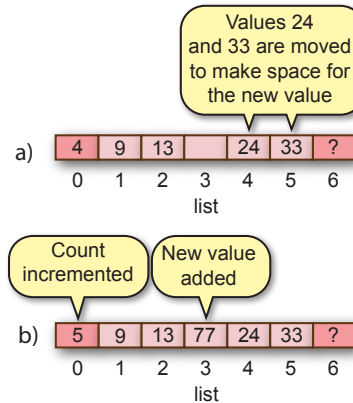
#constant SIZE 5
dim list[SIZE]

rem *** Add sentinel value ***
list[0] = -99
**** Repeat until quit selected ***
SetUpButtons()
repeat
  **** Display menu ***
  Print("1 - Enter value")
  Print("2 - Display number of values held")
  Print("3 - Display all values held")
  Print("4 - QUIT")
  **** Get option ***
  Print("Enter option required(1-4)")
  Sync()
  Sleep(4000)
  option = GetButtonEntry()
  while (option < 1 or option > 4)
    Print("Invalid option. Re-enter.")
    Sync()
    Sleep(2000)
    option = GetButtonEntry()
  endwhile
  **** Execute option ***
  select option
  case 1: **** Add a new value to the list ***
    Print("Enter value to be added : ")
    Sync()
    Sleep(2000)
    value = GetButtonEntry()
    rem *** IF list not full ***
    if list[SIZE] <> -99
      rem *** Search for sentinel ***
      post = 0
      while list[post] <> -99
        inc post
      endwhile
      rem *** Insert new value... ***
      list[post] = value
      rem ***...followed by sentinel ***
      list[post+1] = -99
    else
      Print("List is full")
    endif
    Sync()
    Sleep(2000)
  endcase
  case 2: **** Display the number of items in
    the list ***
    rem *** Search for sentinel ***
    post = 0
    while list[post] <> -99
      inc post
    endwhile
    Print("The list contains "+Str(post)
    + " entries")
    Sync()
    Sleep(2000)
  endcase
  case 3: **** Display the contents of the
    list ***
    if (list[0] = -99)
      Print("The list is empty")
    else
      Print("Values held are")
      post = 0
      while list[post] <> -99
        PrintC(Str(list[post])+" ")
        inc post
      endwhile
    endif
    Sync()
    Sleep(2000)
  endcase
  case 4: **** Quit program ***
    Print("Quitting program in 2 seconds")
    Sync()
  endcase
endrepeat

```

Activity 10.9

A new value could be placed in cells 1, 2, 3, 4, or 5.



Activity 10.10

Modified version of DataCount:

```

#include "Buttons.agc"

#constant SIZE 6
dim list[SIZE]

**** Repeat until quit selected ***
SetUpButtons()
repeat
  **** Display menu ***
  Print("1 - Enter value")
  Print("2 - Display number of values held")
  Print("3 - Display all values held")
  Print("4 - QUIT")
  **** Get option ***
  Print("Enter option required(1-4)")
  Sync()
  Sleep(4000)
  option = GetButtonEntry()
  while (option < 1 or option > 4)
    Print("Invalid option. Re-enter.")
    Sync()
    Sleep(2000)
    option = GetButtonEntry()
  endwhile
  **** Execute option ***
  select option
  case 1: **** Add a new value to the list ***
    Print("Enter value to be added : ")
    Sync()
    Sleep(2000)
    value = GetButtonEntry()
    if list[0] < SIZE
      rem *** Get insert position ***
      Print("Enter position")
      Sync()
      Sleep(1000)
      post = GetButtonEntry()
      while post < 1 or post > list[0]+1
        Print("Position must be between
        1 and "+Str(list[0]+1))
        Sync()
        Sleep(1000)
        post = GetButtonEntry()
      endwhile
      rem *** Make space for new value ***
      for c = list[0] to post step -1
        list[c+1] = list[c]
      next c
      rem *** Increment count ***
      inc list[0]
      rem *** Insert new value ***
      list[post] = value
    else
      Print("List is full")
    endif
  endcase
  case 2: **** Display the number of values held ***
    Print("Number of values held: "+Str(list[0]))
  endcase
  case 3: **** Display all values held ***
    Print("Values held are:")
    for c = 0 to list[0]
      PrintC(Str(list[c])+" ")
    next c
  endcase
  case 4: **** Quit program ***
    Print("Quitting program in 2 seconds")
    Sync()
  endcase
endrepeat

```

```

    else
        Print("List is full")
    endif
    Sync()
    Sleep(2000)
endcase
case 2: /*** Display the number of items in
        the list ***
        Print("The list contains "+Str(list[0])
        + " entries")
        Sync()
        Sleep(2000)
endcase
case 3: /*** Display the contents of the
        list ***
        if (list[0] = 0)
            Print("The list is empty")
        else
            Print("Values held are")
            for c = 1 to list[0]
                PrintC(Str(list[c])+" ")
            next c
        endif
        Sync()
        Sleep(2000)
endcase
case 4: /*** Quit program ***
        Print("Quitting program in 2 seconds")
        Sync()
endcase
endselect
until option = 4
Sleep(2000)
end

```

Activity 10.11

Code for *TopScores*:

```

#include "Buttons.agc"

#constant SIZE 5

dim list[SIZE+1]

/*** Repeat until quit selected ***
SetUpButtons()
repeat
    /*** Display menu ***
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - QUIT")
    /*** Get option ***
    Print("Enter option required(1-4)")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 4)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
    /*** Execute option ***
    select option
    case 1:/*** Add a new value to the list ***
        rem *** Get new score ***
        Print("Enter new score")
        Sync()
        Sleep(1000)
        newscore = GetButtonEntry()
        rem *** Find insertion point ***
        post = 1
        while post <= SIZE and list[post] >=
        newscore
            inc post
        endwhile
        rem *** Create space for new score ***
        for current = SIZE to post step -1
            new = current + 1
            list[new] = list[current];
        next current
        rem *** Add new new score ***
        list[post] = newscore;
        rem *** Increment count ***
        if list[0] < SIZE
            inc list[0]

```

```

        endif
    endcase
case 2: /*** Display the number of items in
        the list ***
        Print("The list contains "+Str(list[0])
        + " entries")
        Sync()
        Sleep(2000)
endcase
case 3: /*** Display the contents of the
        list ***
        if (list[0] = 0)
            Print("The list is empty")
        else
            Print("Values held are")
            for c = 1 to list[0]
                PrintC(Str(list[c])+" ")
            next c
        endif
        Sync()
        Sleep(2000)
endcase
case 4: /*** Quit program ***
        Print("Quitting program in 2 seconds")
        Sync()
endcase
endselect
until option = 4
Sleep(2000)
end

```

Activity 10.12

Modified code for *DataCount*:

```

#include "Buttons.agc"

#constant SIZE 5
dim list[SIZE+1]

/*** Repeat until quit selected ***
SetUpButtons()
repeat
    /*** Display menu ***
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - Delete from position")
    Print("5 - QUIT")
    /*** Get option ***
    Print("Enter option required(1-5)")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 5)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
    /*** Execute option ***
    select option
    case 1:/*** Add a new value to the list ***
        Print("Enter value to be added : ")
        Sync()
        Sleep(2000)
        value = GetButtonEntry()
        if list[0] < SIZE
            rem *** Get insert position ***
            Print("Enter position")
            Sync()
            Sleep(1000)
            post = GetButtonEntry()
            while post < 1 or post > list[0]+1
                Print("Position must be between
                and "+ Str(list[0]+1))
                Sync()
                Sleep(1000)
                post = GetButtonEntry()
            endwhile
            rem *** Make space for new value ***
            for c = list[0] to post step -1
                list[c+1] = list[c]
            next c
            rem *** Increment count ***
            inc list[0]
            rem *** Insert new value ***
            list[post] = value
        else

```

```

        Print("List is full")
    endif
    Sync()
    Sleep(2000)
endcase
case 2: /*** Display the number of items in
        the list ***/
    Print("The list contains "+Str(list[0])
        + " entries")
    Sync()
    Sleep(2000)
endcase
case 3: /*** Display contents of list ***/
    if (list[0] = 0)
        Print("The list is empty")
    else
        Print("Values held are")
        for c = 1 to list[0]
            PrintC(Str(list[c])+" ")
        next c
    endif
    Sync()
    Sleep(2000)
endcase
case 4: /*** Delete from a specified
        position ***/
    rem *** Get position ***
    Print("Enter position of item to be
        deleted")
    Sync()
    Sleep(1000)
    post = GetButtonEntry()
    while post < 1 or post > list[0]
        Print("The position is invalid.
            Re-enter.")
        Sync()
        Sleep(1000)
        post = GetButtonEntry()
    endwhile
    rem *** Delete entry ***
    for current = post+1 to list[0]
        list[current-1] = list[current]
    next current
    rem *** Reduce count ***
    dec list[0]
endcase
case 5: /*** Quit program ***/
    Print("Quitting program in 2 seconds")
    Sync()
endcase
endselect
until option = 5
Sleep(2000)
end

```

Activity 10.13

Modified code for *DataCount*:

```

#include "Buttons.agc"

#constant SIZE 5
dim list[SIZE+1]

/*** Repeat until quit selected ***/
SetUpButtons()
repeat
    /*** Display menu ***/
    Print("1 - Enter value")
    Print("2 - Display number of values held")
    Print("3 - Display all values held")
    Print("4 - Delete value")
    Print("5 - QUIT")
    /*** Get option ***/
    Print("Enter option required(1-5)")
    Sync()
    Sleep(4000)
    option = GetButtonEntry()
    while (option < 1 or option > 5)
        Print("Invalid option. Re-enter.")
        Sync()
        Sleep(2000)
        option = GetButtonEntry()
    endwhile
    /*** Execute option ***/
    select option
        case 1: /*** Add a new value to the list ***/
            Print("Enter value to be added : ")
            Sync()

```

```

Sleep(2000)
value = GetButtonEntry()
if list[0] < SIZE
    rem *** Get insert position ***
    Print("Enter position")
    Sync()
    Sleep(1000)
    post = GetButtonEntry()
    while post < 1 or post > list[0]+1
        Print("Position must be between
            and " + Str(list[0]+1))
        Sync()
        Sleep(1000)
        post = GetButtonEntry()
    endwhile
    rem *** Make space for new value ***
    for c = list[0] to post step -1
        list[c+1] = list[c]
    next c
    rem *** Increment count ***
    inc list[0]
    rem *** Insert new value ***
    list[post] = value
else
    Print("List is full")
endif
Sync()
Sleep(2000)
endcase
case 2: /*** Display the number of items in
        the list ***/
    Print("The list contains "+Str(list[0])
        + " entries")
    Sync()
    Sleep(2000)
endcase
case 3: /*** Display contents of list ***/
    if (list[0] = 0)
        Print("The list is empty")
    else
        Print("Values held are")
        for c = 1 to list[0]
            PrintC(Str(list[c])+" ")
        next c
    endif
    Sync()
    Sleep(2000)
endcase
case 4: /*** Delete value ***
        /*** Enter value to be deleted ***/
        Print("Enter value to be deleted")
        Sync()
        Sleep(1000)
        value = GetButtonEntry()
        /*** Search for value in list ***
        post = 1
        while (post <= list[0] and list[post] <>
            value)
            inc post
        endwhile
        rem *** IF match found, delete entry ***
        if list[post] = value
            rem *** Delete entry ***
            for current = post+1 to list[0]
                list[current-1] = list[current]
            next current
            rem *** Reduce count ***
            dec list[0]
        endif
    endcase
case 5: /*** Quit program ***/
    Print("Quitting program in 2 seconds")
    Sync()
endcase
endselect
until option = 5
Sleep(2000)
end

```

Activity 10.14

No solution required.

Activity 10.15

Modified code for *Shuffle*:

```
#constant SIZE 52
dim list[SIZE]

rem *** Set up values in array ***
for c = 1 to SIZE
    list[c] = c-1
next c
Print("")
rem *** Shuffle ***
for c = 1 to SIZE *20
    rem *** Generate two subscript values ***
    sub1 = Random(1,SIZE)
    sub2 = Random(1,SIZE)
    rem *** Swap values at these positions ***
    temp = list[sub1]
    list[sub1] = list[sub2]
    list[sub2] = temp
next c
rem *** Display shuffled items ***
dim values$(13)={"","Ace","2","3","4","5","6","7",
    "8","9","10","Jack","Queen","King"}
dim suits$(4)={"","Hearts","Diamonds","Spades",
    "Hearts"}
Print("First six cards")
for c = 1 to 6
    PrintC(values$(list[c] mod 13 + 1))
    PrintC(" of ")
    Print(suits$(list[c] / 13 + 1))
next c
Sync()
do
loop
```

The three display statements could even be combined into a single line:

```
Print(Str(values$(list[c] mod 13 + 1))+ " of " +
    Str(suits$(list[c] / 13 + 1)))
```

```
PrintC(Str(scores[row,col])+ " ")
next col
Print("")
next row
Sync()
do
loop
```

Activity 10.16

Code for *DynamicArray*:

```
rem *** Decide size of array ***
size = Random(5,12)
rem *** Set up array ***
dim list[size]
rem *** Store a value in each cell ***
for c = 0 to size
    list[c] = Random(1,20)
next c
rem *** Display the contents of the array ***
for c = 0 to size
    Print(list[c])
next c
Sync()
do
loop
```

Activity 10.17

- a) `dim matrix[2,13]`
- b) `dim matrix [4,1]`
- c) `dim list[7]` This is a one-dimensional array

Activity 10.18

Code for *Using2DArrays*:

```
rem *** Set up array ***
dim scores[4,5]
rem *** Store values in arrays ***
for row = 0 to 4
    for col = 0 to 5
        scores[row,col] = Random(1,20)
    next col
next row
for row = 0 to 4
    for col = 0 to 5
```

Data Types and Operators

In this Chapter:

- ☐ The `dword` Data Type
- ☐ Record Structures
- ☐ Nested Records
- ☐ Arrays of Records
- ☐ Using Other Number Bases
- ☐ Shift Operators
- ☐ Bitwise Boolean Operators

Data Storage

Introduction

The type of every variable we have used so far has been determined by that variable's name. If it ended with a dollar sign, we had a string variable; a hash symbol at the end meant that we had a variable capable of storing a real number; all others were, by default, integer variables. However, as we will see, AGK BASIC allows variables of many other types as well.

Declaring Variables

Any variable used in an AGK BASIC program can be declared. That is, rather than having a variable suddenly appearing for the first time as part of an assignment statement, we can write a line of code stating the variable's name and the type of value it is designed to hold. For example, we can declare *total* as an integer variable using the line:

```
total as integer
```

But, why should we go to the trouble of adding an extra line in order to declare a variable, when we can quite happily get by without doing so? As a general rule, it is considered a good thing to declare your variables. This way we can see the names of every variable in a program or routine by just looking at the section where they are declared. Also we can add a comment beside each variable detailing what that variable is used for (though we should be able to gather that from the name given to the variable). However, there are two other advantages that we derive from declaring our variables. The first of these is that string and real variables no longer need to end with special characters. For example by including the line

```
name as string
```

in a program, we have created a variable called *name* which does not end with a dollar sign, and yet can hold a string value. This allows us to write a line such as

```
name = "Elizabeth"
```

Secondly, by declaring a variable, we can use the type *dword* which, although it cannot store negative values, does allow larger positive integer values than can be placed in a normal integer variable. All the data types available and the range of values they can store are shown in FIG-11.1.

FIG-11.1

Data Types

Data Type	Bytes	Range	
dword	4	0	to 4294967295
integer	4	-2147483648	to 2147483647
float	4	3.4E +/- 38 (7 digits)	
string	NA	NA	

Notice that real variables are declared using the word *float*, not *real*. The new type is *dword* which allows only non-negative values to be stored, but can store double the value that can be reached with a standard *integer*. When declaring variables, the general form of the statement is as shown in FIG-11.2

FIG-11.2

Declaring Variables

```
variable as type
```

where:

variable is the name to be given to the variable.

type is the variable's data type chosen from those given in FIG-11.1.

Type Definitions

When we need to gather a group of related data values, say the name and score of a game player, we can create two separate variables to store this information, as in the lines:

```
name as string
score as integer
```

But because these two pieces of information relate to the one person, it would be better to bind them together in some fashion. AGK BASIC allows us to do this by defining a **record structure**. In fact, the term *record* simply means a collection of related information.

The type Definition

There are two stages to creating a record. Firstly, we must start by defining a blueprint for the structure we require. This is done using a **type** statement. For the data described above this would be coded as:

```
type ContestantType
  name as string
  score as integer
endtype
```

Notice that the keyword **type** is followed by an identifying name, *ContestantType*. This is the name we wish to give to this record structure design. We can choose any name as long as it conforms to the same rules as naming variables in our programs, but it's a good idea to end the name with *Type* - that way we'll remember that it is not a variable name when we see it used in the program.

The parts that make up a structure are known as **fields**. So the *ContestantType* structure contains fields called *name* and *score*. Fields can also be defined without explicitly declaring each field's type. So it is quite acceptable to write

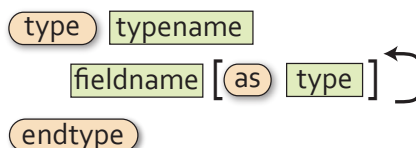
```
type ContestantType
  name$
  score
endtype
```

although in the above case, we are forced to add the \$ symbol to indicate a string field.

There is no restriction to the number of fields that can be named within a type definition, but it cannot contain arrays. A type definition has the format shown in FIG-11.3.

FIG-11.3

Declaring Records



where:

typename is the name to be given to the type being defined.

fieldname is the name of a field within the structure.

type is the data type of the field (from FIG-11.1).

As many **type** declarations as required can be placed in a single program.

Activity 11.1

Start a new project called *UsingRecords*.

After deleting the default contents of *main.agc*, add a **type** definition called *DateType* for a record containing the fields *day*, *month* and *year*, all of which are of type integer.

Save your project.

Declaring Variables of a Defined Type

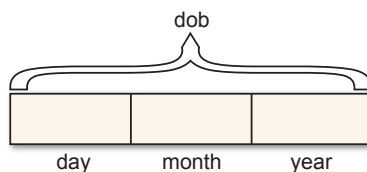
Once we have created the **type** definition, we can then create variables of this type using the **as** statement. For example:

```
dob as DateType
```

The variable *dob* is now created containing all the fields defined in *DateType* (see FIG-11.4).

FIG-11.4

DateType's Structure



If we need several variables of the same type, we need an **as** line for each variable created:

```
dob          as DateType
anniversary  as DateType
wifedob      as DateType
```

The variables constructed in this way are referred to as **records** or **composite variables**.

Activity 11.2

In *UsingRecords*, create two variables, *p1* and *p2* of type *DateType*.

Save your project.

Accessing the Fields in a Composite Variable

When we want to access the fields in a record, we need to start with the variable name followed by a full-stop and then the field name. So, having declared a record variable with the line


```
challenger as ContestantType
```

we can access that variable's *score* field using the term:

```
challenger.score
```

As long as we use the correct term, then we can do anything with a record's field that we might do with a standard variable. Hence, all of the following are valid statements:

```
challenger.name = "Liz Heron"  
Print(challenger.score)  
if challenger.score > 1000  
    Print("High score")  
endif
```

Activity 11.3

In *UsingRecords*, set the fields in *p1* to the date 22/11/1963 and those in *p2* to 14/9/1979.

Add two `Print()` statements which display the dates held in *p1* and *p2*.

Turn your code into a complete program and check that it operates as expected.

Resave your project.

In most cases, it is invalid to try to treat a record as a single entity. For example, the line

```
Print(challenger)
```

is not allowed. Instead, the individual fields must be displayed separately:

```
Print(challenger.name)  
Print(challenger.score)
```

The same restriction is true for all other statements - except one. It is allowable to copy the contents of one record into a second record of the same type with just a single statement. For example, if we define two records:

```
champion    as ContestantType  
challenger  as ContestantType
```

and assign values to one of the records:

```
challenger.name = "Liz Heron"  
challenger.score = 781
```

then we can copy all the values in *challenger* into *champion* with the single line:

```
champion = challenger
```

Nested Record Structures

In the last chapter we created an array of top times, but this held only the actual times themselves. In a video game, the name of the player associated with each score would also be held.

To do this we start with defining a record type:

```
type PlayerType
    name      as string
    score     as integer
endtype
```

But if we wanted to store the date on which a winning time was achieved, then we could add a date field to this structure:

```
type PlayerType
    name      as string
    score     as integer
    achieved  as DateType
endtype
```

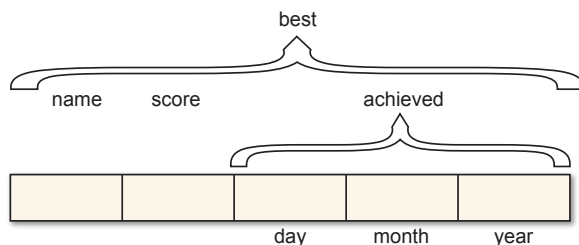
Now we have, within one record structure (*PlayerType*), a field called *achieved*, which is itself a record structure (*DateType*). So, if we declare a variable of this type with the line

```
best as PlayerType
```

then the data structure created is given visual form in FIG-11.5.

FIG-11.5

Nested Structures



This setup is known as a **nested records** structure.

If we create a variable of this type:

```
player as PlayerType
```

we can access the *name* and *score* fields in the usual way:

```
player.name
player.score
```

but the last field, *achieved*, being a record structure in its own right, must be accessed a field at a time:

```
player.achieved.day
player.achieved.month
player.achieved.year
```

Activity 11.4

Start a new project called *NestedRecords* and define a record structure, *TimeType*, which has two fields: *minutes* and *seconds*, both of which are of type integer. Define a second record structure, *BestType*, which contains two fields: *name* and *time*. *name* is of type **string** and *time* of type *TimeType*. Declare a variable, *winner*, of type *BestType* and set its contents to *Emily Knight*, 2 minutes 31 seconds. Display the contents of the record.

It is not possible in AGK BASIC to create an array as one of the fields within a record structure.

Arrays of Records

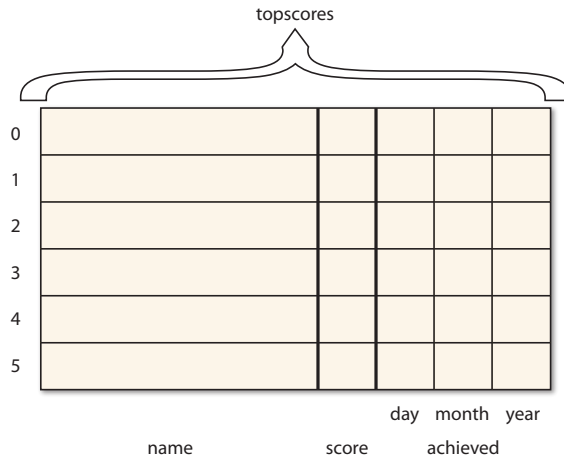
Although we cannot place an array inside a **type** definition, we can create an array of records. For example, we could use the line

```
dim topscores[5] as PlayerType
```

to create the array shown visually in FIG-11.6.

FIG-11.6

An Array of Records



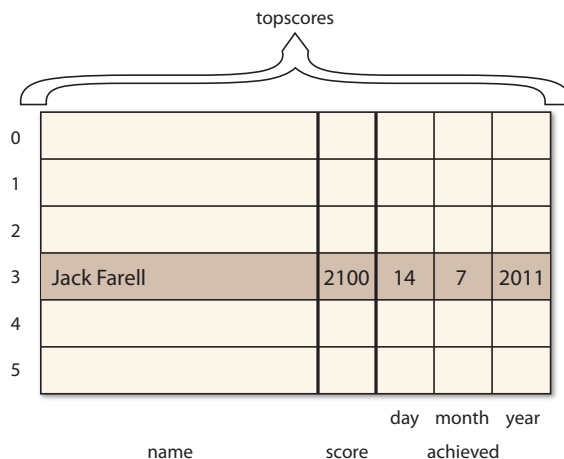
To access a data item in this structure, we start by specifying the array name and element number. This is followed by a full stop and the name of the field to be accessed. So, assuming we're not using element zero, we could set the third entry in the array to *Jack Farrell, 2100, 14/7/2011* with the lines:

```
topscores[3].name = "Jack Farrell"  
topscores[3].score = 2100  
topscores[3].achieved.day = 14  
topscores[3].achieved.month = 7  
topscores[3].achieved.year = 2011
```

The effect of these five statements is shown in FIG-11.7.

FIG-11.7

Storing a Value in an Array of Records



Activity 11.5

Modify *NestedRecords* so that the variable *winner* is replaced by an array of six elements called *winners* of type *BestType*.

The program should then assign random values to each field in records *winners[1]* to *winners[5]* and display all of those values. Make use of the *RandomString()* function in *StringLibrary.agc* to create values for the *name* fields.

Test and save your project.

Summary

- Variables can be specifically declared.
- Declared variables can be of type `dword` as well as `integer`, `float` and `string`.
- Declared string variables need not have names that end with a dollar (\$) symbol.
- Real variables are declared as `float`.
- Declared real variables need not have names that end with the # symbol.
- `type` definitions can be used to create record structures.
- The data items defined within a `type` are known as fields of the record.
- Fields cannot be arrays.
- Once a `type` has been defined, variables of that type can be declared.
- Variables of a defined type are known as record variables.
- Fields within a record variable are accessed using the format

`record_variable_name.field_name`

- Arrays of a defined type can also be declared.
- To access a field in a record within an array use the format

`array_name[subscript].field_name`

Data Manipulation

Introduction

Earlier in the book we looked at the basic arithmetic operators such as addition and multiplication. There are, however, several other operators that perform more specialised tasks.

Other Number Systems

A PDF file (*NumbersSystems.pdf*), available in *AGKDownloads/Chapter11*, gives a greater description of the various number systems mentioned here.

Although we spend most of our time working in the decimal number system (properly called the denary number system), computers work in binary, where every possible number (and character) is represented as a pattern of 1s and 0s.

If we want to assign a specific binary value to an integer variable, then we can do so by starting our number with a percent (%) sign. For example, the lines

```
value = %01000001
```

will store the binary value 01000001 in the variable *value*. If we are good at converting between number systems, we could have achieved the same effect by writing

```
value = 65
```

A specific hexadecimal value can also be assigned by starting with 0X (zero X) as in the line:

```
value = 0XFF
```

The more obscure octal number base is also possible by starting the value with 0C (zero C) as in:

```
value = 0C53
```

Remember, no matter what number base we use in our program code, this makes no difference to how the values are stored. They are always stored in binary. The different number bases are available to make things easier for us - not the computer.

The program in FIG-11.8 uses all four number systems to assign values to four variables. The contents of the variables are then displayed.

FIG-11.8

Using Different
Number Bases

```
REM *** Assign a value to four variables ***
REM *** using a different number base for each ***
v1 = 65
v2 = %01000001
v3 = 0X41
v4 = 0C101
REM *** Display the contents of each variable ***
Print(v1)
Print(v2)
Print(v3)
Print(v4)
Sync()
do
loop
```

Activity 11.6

Create a new project called *NumberBases* and implement the code given in FIG-11.8.

What four values are displayed by the program?

Modify the program so that *v2* is displayed in binary and *v3* in hexadecimal.

Save the project.

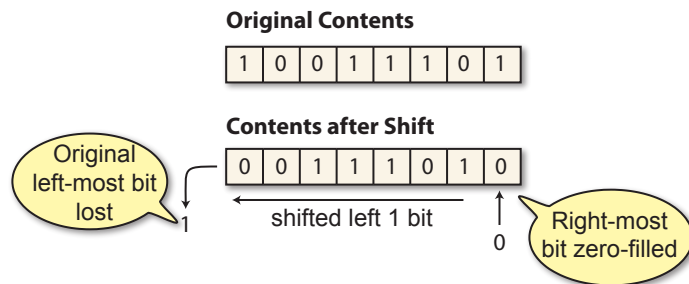
Shift Operators

If you have ever sat in a well-organised show and been asked to move all the way along the row, you'll have an idea of what shifting is all about.

Shift operators allow the contents of an integer variable (*integer* or *dword*) to be moved. For example, in FIG-11.9 we see the contents of a single byte being moved one place to the left.

FIG-11.9

Shift Left

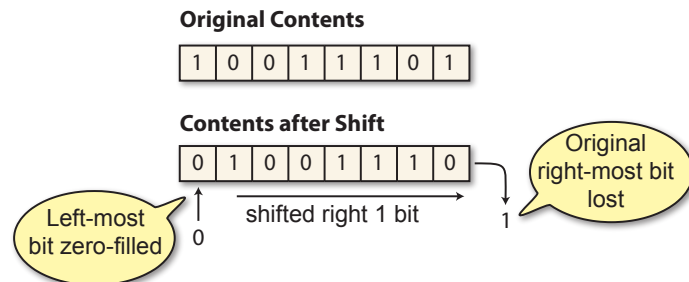


We can see from the diagram that the bit that starts out as the left-most bit falls off the end and is lost. Also, to fill the gap created on the right-hand side of the byte a new digit (a zero) is inserted.

A similar effect occurs when bits are shifted to the right (see FIG-11.10).

FIG-11.10

Shift Right



Activity 11.7

Write down the result of the following shifts:

- a) 10110001 shifted 2 places to the left
- b) 10110001 shifted 3 places to the right

So why would we want to shift the contents of a variable? Well, one reason is that left shifting is a very efficient way of multiplying by powers of 2.

In the decimal system, if we start with a number (say 12) and shift the digits one column to the left (120), we've multiplied the original value by 10; if we shift the original value 2 places to the left (1200), we have multiplied the value by 100.

When we do the same thing to a binary number (say, 00001100) and shift it 1 place to the left (00011000), the original value has been multiplied by 2; a shift of 2 places to the left (00110000) would multiply the original value by 4.

Conversely, a single right shift divides a value by 10 in the decimal system (200 becomes 20) and by 2 in the binary system (00000100 becomes 00000010).

The Shift Left Operator (<<)

If we set up a variable containing the number 7 (00000111) with the lines

```
value = 7
```

then we can shift the contents of *value* one place to the left using the shift-left operator (<<) with the line

```
value = value << 1
```

and hence, since the computer holds everything in binary, the number held in *value* is doubled to 14 (00001110).

FIG-11.11

The Shift Left
Operator

The shift operator is used in an expression which has the form shown in FIG-11.11.



where:

- | | |
|---------------|--|
| ivalue | is the integer-type variable, constant or expression whose value is to be left-shifted. |
| ibits | is an integer value representing the number of places the contents are to be shifted. For integer values, which are stored over 32 bits there is no purpose in this value being greater than 32; dword variables can be shifted up to 64 positions. |

For example, the line

```
Print(%00111010 << 2)    rem *** 00111010  is  58 ***
```

would display 232 (which is 11101000 in binary).

Activity 11.8

Create a new project, *Bits01*, which sets an integer variable *num* to binary 11 and shifts the contents of *num* 1 place to the left.

The value held in *num* should be displayed both before and after the shift. Also, display the before and after value in both decimal and binary.

The Shift Right Operator (>>)

By shifting the contents of an integer variable to the right (using the shift-right operator >>) we halve the numeric value of that variable. Shifting two places results in a quartering of the original value. For example, the lines

```
num2 = 24
Print(num2 >> 2)
```

will display the value 6 (a quarter of 24). Of course, if we halve an odd number by using left shift, then the fraction will be lost. So,

```
num = 13
Print(num >> 1)
```

would display the value 6.

Activity 11.9

Modify *Bits01* so that the value in *num* is shifted one place to the right rather than the left.

Display the value before and after the shift in decimal and binary.

Bitwise Boolean Operators

We met the Boolean operators **and**, **or** and **not** in an earlier chapter. These combined true and false values to give an overall true or false result. Bitwise Boolean operators work on the individual bits that make up a value, with a binary 1 being treated as the equivalent of true and zero being false.

The Bitwise NOT Operator (..)

The bitwise NOT operator, **..**, changes all 1s in a value to 0s and changes all 0s to 1s. The operator has a rather strange syntax (see FIG-11.12).

FIG-11.12

The Bitwise NOT Operator



where:

ivalue is an integer value whose bits are to be complemented.

0 is the numeric value zero. Actually, any integer value can be used as this has absolutely no effect on the result and is only required to satisfy the syntax of the operator.

Therefore, if we start with the line

```
num = %10011101
```

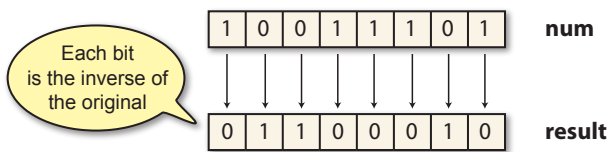
then the statement

```
result = num..0
```

uses the bitwise NOT operator to create a binary value whose bits are the exact complement of those in *num* and stores the value in *result* (see FIG-11.13).

FIG-11.13

Bitwise NOT



Of course we can do all of this in decimal (or even hexadecimal or octal) and get exactly the same value:

```
num = 15
num = num..0
```

We can even display the contents of result in binary with the line:

```
Print(Bin(num))
```

Activity 11.10

Create a new project, *Bits02*, which performs the following logic:

Create integer variables *s* and *f*
 Assigns *s* the binary value 10101001
 Set *f* = bit complement of *s*
 Display *s* and *f* in binary

Test and save your project.

The Bitwise AND Operator (&&)

The individual bits of two values can be ANDed together using the bitwise AND operator, &&.

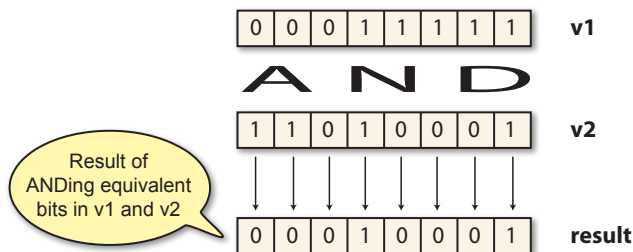
The code

```
v1 = %00011111
v2 = %11010001
result = v1 && v2
Print(Bin(result))
```

displays the value 00010001. How this result is derived is shown in FIG-11.14.

FIG-11.14

The Bitwise AND Operator



Activity 11.11

Start a new project, *Bits03*, and create a program based on the code above that ANDs the values 00011111₂ and 11010001₂.

Check that the expected result is displayed.

The Bitwise OR Operator (||)

The individual bits of two values can be ORed together using the bitwise OR operator, ||.

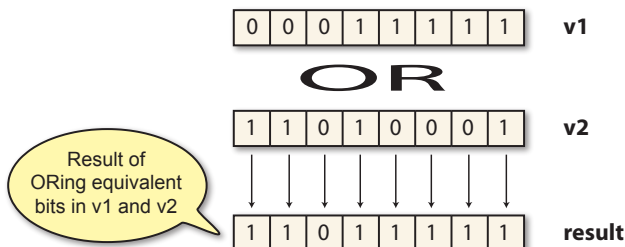
The code

```
v1 = %00011111
v2 = %11010001
result = v1 || v2
Print(BIN(result))
```

displays the value 11011111. How this result is derived is shown in FIG-11.15.

FIG-11.15

The Bitwise OR Operator



Activity 11.12

Modify *Bits03* to OR (rather than AND) the two values *v1* and *v2*.

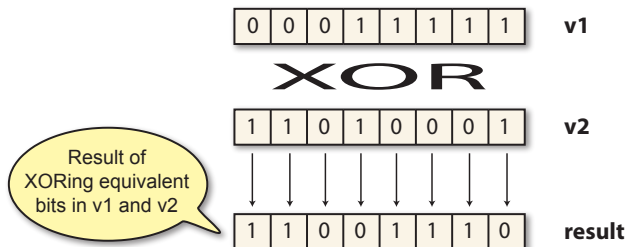
Check that the expected results are displayed.

The Bitwise Exclusive OR Operator (~)

The exclusive OR operation - sometimes written as XOR - returns a result of 1 if the two bits being compared are different and a 0 if they are the same (see FIG-11.16).

FIG-11.16

The Bitwise XOR Operator



Activity 11.13

Modify *Bits03* to XOR the two values *v1* and *v2*.

Check that the expected results are displayed.

Activity 11.14

Work out the results of the following operations:

- a) 01100111 ..0
- b) 01010011 && 10000110
- c) 01000011 || 00010000
- d) 00100111 ~ 01001101

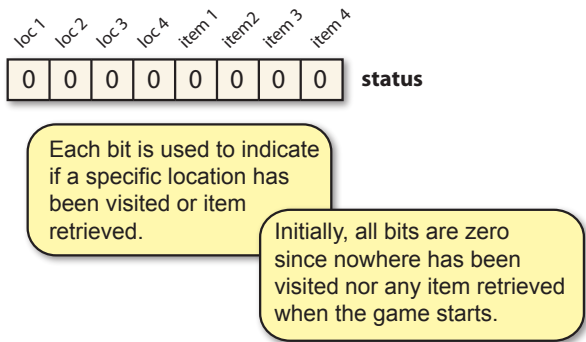
A Practical Use For Bitwise Operations

Imagine we are writing an adventure game which contains four locations and 4 items that the player might pick up on his journey. When the player reaches the end of the game, he wins only if he has visited location 1 and possesses items 2 and 3.

To finish the game correctly, the program will have to keep track of where the player has been and what he has picked up. Perhaps we could use 8 variables, or an 8-element array to do this. Initially, set everything to zero and then when a place is visited or an item taken, set that variable (array element) to 1. At the end of the game check to see if the appropriate variables are set to 1 and our problem is solved. This approach is fine but can be wasteful of memory, especially if the real game has thousands of locations and hundreds of items.

Another approach is to use a single bit for each piece of information - after all we only need to store a 0 or 1 in each case. Since we need to record 8 pieces of information, we can use a single byte. All we have to do is decide the purpose of each bit in that byte (see FIG-11.17).

FIG-11.17
Using Bits to Record Information



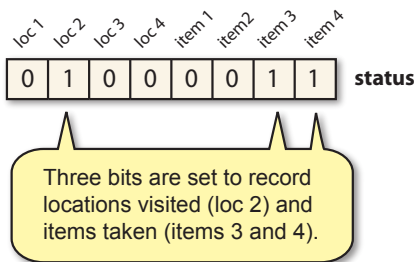
We can create this initial setup with the lines:

```
status = 0
```

Since *status* is an integer, it will be assigned 32 bits, but we will only be using the right-most 8 bits of this space.

Now, let's say the player visits location 2 and picks up items 3 and 4, this means we need to change *status* as shown in FIG-11.18).

FIG-11.18
Recording Locations and Items



Setting these bits requires the code:

```
status = %01000011
```

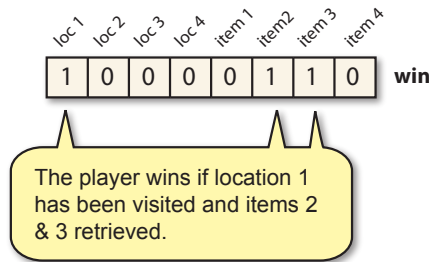
Later, the player visits location 4, so *status* changes again:

```
status = status || %00010000
```

When we come to the end of the game, we can check if all the criteria for winning have been met by setting up a variable (*win*) to reflect the winning condition (see FIG-11.19).

FIG-11.19

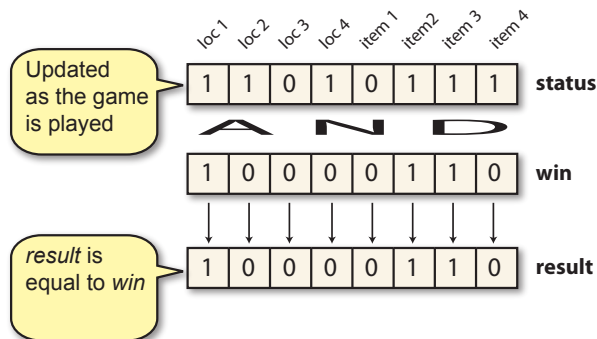
The Win Requirements



Now, all we have to do is use the bitwise and operation on *status* and *win* and check that *result* has the same value as *win* (see FIG-11.20).

FIG-11.20

Checking for a Win



This is coded as:

```
#constant win %10000110
result = status && win
if result = win
    Print("You win")
endif
```

Summary

- Integer constants can be specified in binary, hexadecimal or octal.
- Binary constants begin with the % symbol.
- Hexadecimal constants begin with 0X.
- Octal constants begin with 0C.
- Bits within an integer variable can be shifted to the left using the shift left operator (<<).
- Bits within an integer variable can be shifted to the right using the shift right operator (>>).
- The bits of an integer value can be complemented (NOTed) using the bitwise NOT operator (~).
- The bits of two integer values can be ANDed using the bitwise AND operator (&&).

- The bits of two integer values can be ORed using the bitwise OR operator (`||`).
- The bits of two integer values can be XORed using the bitwise XOR operator (`⊕`).

Solutions

Activity 11.1

Code for *UsingRecords*:

```
type DateType
  day
  month
  year
endtype
```

You have the option to add

```
as integer
```

after each field name.

Activity 11.2

Code for *UsingRecords*:

```
type DateType
  day
  month
  year
endtype
```

```
p1 as DateType
p2 as DateType
```

Activity 11.3

Code for *UsingRecords*:

```
type DateType
  day
  month
  year
endtype
```

```
p1 as DateType
p2 as DateType
```

```
rem *** Assign values ***
p1.day = 22
p1.month = 11
p1.year = 1963
```

```
p2.day = 14
p2.month = 9
p2.year = 1979
```

```
rem *** Display dates ***
Print(Str(p1.day)+"/"+Str(p1.month)+"/"+
  ⚡Str(p1.year))
Print(Str(p2.day)+"/"+Str(p2.month)+"/"+
  ⚡Str(p2.year))
Sync()
do
loop
```

Activity 11.4

Code for *NestedRecords*:

```
type TimeType
  minutes
  seconds
endtype
```

```
type BestType
  name as string
  time as TimeType
endtype
```

```
winner as BestType
```

```
rem *** Assign Values ***
winner.name = "Emily Knight"
winner.time.minutes = 2
winner.time.seconds = 31
```

```
rem *** Display value ***
```

```
Print(winner.name+" "+Str(winner.time.minutes)+
  ⚡mins "+Str(winner.time.seconds)+" secs")
Sync()
do
loop
```

Activity 11.5

Code for *NestedRecords*:

```
#include "StringLibrary.agc"
```

```
type TimeType
  minutes
  seconds
endtype
```

```
type BestType
  name as string
  time as TimeType
endtype
```

```
dim winners[5] as BestType
```

```
rem *** Assign Values ***
for c = 1 to 5
  winners[c].name = RandomString(20)
  winners[c].time.minutes = Random(1,3)
  winners[c].time.seconds = Random(0,59)
next c
```

```
rem *** Display value ***
for c = 1 to 5
  Print(winners[c].name+" "+Str(winners[c].time.
    ⚡minutes)+" mins "+Str(winners[c].time.
    ⚡seconds)+" secs")
next c
Sync()
do
loop
```

Activity 11.6

The value displayed by all four `Print` statements is 65 since each value assigned is equivalent to 65 in decimal.

Modified code for *NumberBases*:

```
REM *** Assign a value to four variables ***
REM *** using a different number base for each ***
v1 = 65
v2 = %01000001
v3 = 0X41
v4 = 0C101
REM *** Display the contents of each variable ***
Print(v1)
Print(Bin(v2)+" Binary")
Print(Hex(v3)+" Hexadecimal")
Print(v4)
Sync()
do
loop
```

Activity 11.7

- a) 11000100
- b) 00010110

Activity 11.8

Code for *Bits01*:

```
rem *** Using Shift Left Operator ***

rem *** Assign value to variable ***
num = %11
rem *** Display value in dec and bin ***
Print("Before")
Print("Decimal : "+Str(num))
Print("Binary : " +Bin(num))
rem *** Shift left 1 bit ***
num = num << 1
rem *** Display value in dec and bin ***
Print("After")
Print("Decimal : "+Str(num))
Print("Binary : " +Bin(num))
```

```

Sync()
do
loop

```

Activity 11.9

Modified code for *Bits01*:

```

rem *** Using Shift Right Operator ***

rem *** Assign value to variable ***
num = %11
rem *** Display value in dec and bin ***
Print("Before")
Print("Decimal : "+Str(num))
Print("Binary : " +Bin(num))
rem *** Shift right 1 bit ***
num = num >> 1
rem *** Display value in dec and bin ***
Print("After")
Print("Decimal : "+Str(num))
Print("Binary : " +Bin(num))
Sync()
do
loop

```

Activity 11.10

Code for *Bits02*:

```

rem *** Using the NOT Operator ***

rem *** Assign value to variable ***
s = %10101001
rem *** Calculate NOT s ***
f = s.0
rem *** Display s in dec and bin ***
Print("S:")
Print("Binary : " +Bin(s))
rem *** Display f in dec and bin ***
Print("F:")
Print("Binary : " +Bin(f))
Sync()
do
loop

```

Activity 11.11

Code for *Bits03*:

```

rem *** Using the AND Operator ***

rem *** Assign values to variables ***
v1 = %00011111
v2 = %11010001
rem *** AND both values ***
result = v1 && v2
rem *** Display results ***
Print(Bin(v1)+" AND "+Bin(v2)+" = "+Bin(result))
Sync()
do
loop

```

Activity 11.12

Modified code for *Bits03*:

```

rem *** Using the OR Operator ***

rem *** Assign values to variables ***
v1 = %00011111
v2 = %11010001
rem *** OR both values ***
result = v1 || v2
rem *** Display results ***
Print(Bin(v1)+" OR "+Bin(v2)+" = "+Bin(result))
Sync()
do
loop

```

Activity 11.13

Modified code for *Bits03*:

```

rem *** Using the XOR Operator ***

```

```

rem *** Assign values to variables ***
v1 = %00011111
v2 = %11010001
rem *** XOR both values ***
result = v1 ^^ v2
rem *** Display results ***
Print(Bin(v1)+" XOR "+Bin(v2)+" = "+Bin(result))
Sync()
do
loop

```

Activity 11.14

a) 10011000 (the leading 24 bits will be set to 1) so what you see on the screen is 11111111 11111111 11111111 10011000.

b) 00000010

c) 01010011

d) 01101010

12

File Handling

In this Chapter:

- ☐ Writing Data to a File
- ☐ Reading Data from a File
- ☐ Checking that a File Exists
- ☐ Determining a File's Size
- ☐ Deleting a File
- ☐ Managing Folders

Introduction

If we intend to hold onto information after an app has been completed, then we need to store that information on a backing storage device such as a disk or flash memory.

Typical information will be such things as app settings and game score details.

Accessing Files

We'll start by finding out how to write data to a file. The basic steps involved consist of the main stages:

Open the file for writing
Output data to the file
Close the file

OpenToWrite()

We need to start by creating the new file in which our data is to be held. This can be done using the `OpenToWrite()` statement which has the format shown in FIG-12.1.

FIG-12.1

OpenToWrite()

Version 1

`OpenToWrite (id , file , imode)`

Version 2

`integer OpenToWrite (file , imode)`

where:

- | | |
|--------------|--|
| id | is an integer value giving the ID to be assigned to the file. No two files may have the same ID value while open. |
| file | is a string giving the name of the file to be opened. The name may include path information. |
| imode | is an integer value (0 or 1). If zero, the current contents of the file will be deleted before new data is added. If 1, the new data is added to the end of the existing data within the file. |

A typical statement might be:

```
stdfile = OpenToWrite("fastesttime.std",1)
```

If a file matching the name given does not exist, the file will be created automatically.

The file is assumed to be in the folder used by the device for the app's data files (this is determined by the operating system on the device). If the filename includes folder details, that folder is assumed to be a subfolder of the app data folder. If the folder does not exist, it will be created automatically.

The second version of the statement automatically assigns an ID to the file and returns that value. All subsequent statements that access the file will use this ID.

WriteInteger()

Once a file has been opened for writing, we can then store within it the information we want to save. Values must be written one item at a time. There is a different command for each data type. To write an integer value to the file, we use the `WriteInteger()` statement (see FIG-12.2).

FIG-12.2

WriteInteger()

The diagram shows the syntax for the `WriteInteger()` statement. It consists of the function name `WriteInteger` in an orange rounded rectangle, followed by an opening parenthesis `(` in a light green circle, then the parameter `id` in a light green rectangle, followed by a comma `,` in a light green circle, then the parameter `ivalue` in a light green rectangle, and finally a closing parenthesis `)` in a light green circle.

where:

id is an integer value giving the file ID.

ivalue is an integer value giving the actual data to be written to the file.

WriteFloat()

If the value to be written to the file is a real one (rather than an integer), then the `WriteFloat()` statement is used (see FIG-12.3).

FIG-12.3

WriteFloat()

The diagram shows the syntax for the `WriteFloat()` statement. It consists of the function name `WriteFloat` in an orange rounded rectangle, followed by an opening parenthesis `(` in a light green circle, then the parameter `id` in a light green rectangle, followed by a comma `,` in a light green circle, then the parameter `fvalue` in a light green rectangle, and finally a closing parenthesis `)` in a light green circle.

where:

id is an integer value giving the file ID.

fvalue is the real value to be written to the file.

WriteString()

To write a standard string value to a file the `WriteString()` statement is used (see FIG-12.4).

FIG-12.4

WriteString()

The diagram shows the syntax for the `WriteString()` statement. It consists of the function name `WriteString` in an orange rounded rectangle, followed by an opening parenthesis `(` in a light green circle, then the parameter `id` in a light green rectangle, followed by a comma `,` in a light green circle, then the parameter `string` in a light green rectangle, and finally a closing parenthesis `)` in a light green circle.

where:

id is an integer value giving the file ID.

string is the string value to be written to the file. The string will be terminated using the standard null character.

WriteLine()

To create a string terminated with a return character rather than a null, use the `WriteLine()` statement (see FIG-12.5).

FIG-12.5

WriteLine()

The diagram shows the syntax for the `WriteLine()` statement. It consists of the function name `WriteLine` in an orange rounded rectangle, followed by an opening parenthesis `(` in a light green circle, then the parameter `id` in a light green rectangle, followed by a comma `,` in a light green circle, then the parameter `string` in a light green rectangle, and finally a closing parenthesis `)` in a light green circle.

where:

id is an integer value giving the file ID.

string is the string value to be written to the file. The string will be terminated using a return character.

WriteByte()

There are occasions when it is useful to write to a file one byte at a time. This can be done using the `WriteByte()` statement (see FIG-12.6).

FIG-12.6

WriteByte()

`WriteByte` (`id` , `iv`)

where

id	is an integer value giving the file ID.
iv	is an integer value (one byte in length) to be written to the file. If <i>iv</i> is stored over more than one byte, only the least-significant byte is written.

CloseFile()

Once we've finished writing data, the file must be closed. This frees up RAM space that has been linked to the file and ensures that the last of the data has been written to the file. This is achieved using the `CloseFile()` statement which has the format given in FIG-12.7.

FIG-12.7

CloseFile()

`CloseFile` (`id`)

where:

id	is an integer value giving the ID of the file to be closed.
-----------	---

Once a file is closed it cannot be used again until it is reopened.

This Activity demonstrates how to write the contents of a record to a file.

Activity 12.1

Start a new project, *UseDataFile* and modify *main.agc* to read:

```
rem *** set up record structure ***
type PlayerType
    name as string
    score as integer
endtype
rem *** Create record variable ***
nol as PlayerType
rem *** Assign values to the fields within the records ***
nol.name = "Jane"
nol.score = 613
rem *** Open file ***
myfile = OpenToWrite("Test.dat",0)
rem *** Write record fields to file ***
WriteString(myfile,nol.name)
WriteInteger(myfile,nol.score)
rem *** Close file ***
CloseFile(myfile)
Print("Writing to file completed")
Sync()
do
loop
```

Test and save the program.

OpenToRead()

Sometime after data has been written to a file, you are going to want to read that data back from the file. To do this we need to start by opening the file for reading using the `OpenToRead()` statement which has the format shown in FIG-12.8.

FIG-12.8

`OpenToRead()`

Version 1

`OpenToRead (id , file)`

Version 2

`integer OpenToRead (file)`

where:

id is an integer value giving the ID to be assigned to the file.

file is a string giving the name of the file to be opened. The name may include path information.

The second version statement returns the ID it has assigned to the file.

If the specified file is not found in the default folder used by the target device, the game's *media* file will also be searched. Since each app creates its own data folder, it's important that you use the same program to both write and read a file. If the file cannot be found then the program will abort with an error message. All subsequent statements that access the file will use this ID.

ReadInteger()

When a file has been opened for reading, we must read the information from the file one item at a time. There is a different command for each data type. To read an integer value from the file, we use the `ReadInteger()` statement (see FIG-12.9).

FIG-12.9

`ReadInteger()`

`integer ReadInteger (id)`

where:

id is an integer value giving the file ID.

The statement returns the integer value read from the file.

ReadFloat()

The `ReadFloat()` statement reads a real value from a file (see FIG-12.10).

FIG-12.10

`ReadFloat()`

`float ReadFloat (id)`

where:

id is an integer value giving the file ID.

The statement returns a real value read from the file.

ReadString()

FIG-12.11

ReadString()

The `ReadString()` statement reads a standard string from a file (see FIG-12.11).

string `ReadString` (`id`)

where:

id is an integer value giving the file ID.

The statement returns a null-terminated string read from the file.

ReadLine()

FIG-12.12

ReadLine()

The `ReadLine()` statement, reads a return-terminated string from a file (see FIG-12.12).

string `ReadLine` (`id`)

where:

id is an integer value giving the file ID.

The statement returns a normal string.

ReadByte()

FIG-12.13

ReadByte()

To read a single byte from an existing file, use `ReadByte()` (see FIG-12.13).

integer `ReadByte` (`id`)

where

id is an integer value giving the file ID.

When we have finished reading from a file, the file should be closed using the `CloseFile()` statement.

A short program to read the data stored in file *Test.dat* by Activity 12.1 is shown in FIG-12.14.

FIG-12.14

Reading From a File

➡ This new code must be placed in the same project as the earlier code so that the data file will be found. The data file is held in a folder whose name and position is based on the project's name.

```
rem *** Open file for reading ***

myfile = OpenToRead("Test.dat")
rem *** read data from file ***
name$ = ReadString(myfile)
score = ReadInteger(myfile)
rem *** Close the file ***
CloseFile(myfile)
rem *** Display information read ***
Print(name$)
Print(score)
Sync()
do
loop
```

Activity 12.2

Add the code in FIG-12.12 to *UseDataFile's main.agc*. Place the code at the start of the program. Check that the program correctly reads back the information stored in the file. Save your project.

Notice that it is entirely the responsibility of the programmer to read the correct type of data from the file. When we created *Test.dat*, the data written to it was a string followed by an integer. So, when we read the contents of the same file, we must make sure that we first read a string and then an integer.

FileEOF()

Although *Test.dat* contained only two items of data, a file may contain as many items of data as required. Sometimes we may not even know exactly how many data items have been written to a file. For example, the program in FIG-12.15 generates random numbers between 1 and 12, writing each value to a file. The program stops when a 12 is generated (the 12 is not written to the file).

FIG-12.15

Writing an Unknown
Quantity of Data

```
rem *** Open file for writing ***

myfile = OpenToWrite("Numbers.dat",0)
rem *** Generate random number ***
no = Random(1,12)
rem *** WHILE no not 12 DO ***
while no <> 12
    rem *** Write number to file ***
    WriteInteger(myfile,no)
    rem *** Generate another number ***
    no = Random(1,12)
endwhile
rem *** Close the file ***
CloseFile(myfile)
end
```

Activity 12.3

Start a new project called *LongFile* and set *main.agc* to match the code given in FIG-12.15. Run your project.

It would be better if we could see what values are being written to the file. Modify your code so that a list of the values written to the file are also displayed in the app window.

Test and save your updated project.

Broadcast the project to your tablet or phone and run it there.

Attempting to read all the data from the file we have just created causes a problem because we do not know how many values are in the file. We would like our program to just keep on reading until we reach the end of the data.

This is where the `FileEOF()` statement comes in. This statement checks to see if the end of the file has been reached. The statement has the format shown in FIG-12.16.

FIG-12.16

FileEOF()

integer **FileEOF** ((**id**))

where:

id is an integer value giving the file ID.

The statement returns 1 if the end of file has been reached, otherwise zero is returned.

Typically, the statement is used in a **while** loop:

```

rem *** Read from file ***
while fileEOF(id) = 0
    rem *** Handle value read from file ***
    rem *** read from file ***
endwhile

```

The program in FIG-12.17 makes use of the **FileEOF()** statement to read the file created by the program in FIG-12.15.**FIG-12.17**Using the FileEOF()
Statement

```

rem *** Open file for reading ***
myfile = OpenToRead("Numbers.dat")
rem *** read a value from the file ***
num = ReadInteger(myfile)
rem *** WHILE not EOF DO ***
while fileEOF(myfile) <> 1
    rem *** Display the value read ***
    Print(num)
    rem *** read a value from the file ***
    num = ReadInteger(myfile)
endwhile
rem *** Close the file ***
CloseFile(myfile)
Print("Finished")
Sync()
do
loop

```

Activity 12.4

Add the code in FIG-12.17 to *LongFile's main.agc*. Place this new code at the start of the file. Check that the program correctly reads back the information stored in the file when run from your PC and tablet or phone. Save your project.

File Management

We have looked at the commands necessary for reading and writing to files, but there are a few other file-related commands which will help us manage those files.

GetFileExists()

If you open a file for writing and that file does not exist, the program will automatically create that file, but if we attempt to open a non-existent file for reading then the program will crash. To avoid such a crash, we can use the **GetFileExists()** statement

to check that the file we intend to use does actually exist before executing an `OpenForReading()` statement. The format of the `GetFileExists()` statement is given in FIG-12.18.

FIG-12.18

`GetFileExists()`

integer `GetFileExists` ((`file`))

where:

file is a string giving the name of the file to be checked.

If the named file does exist, the statement returns 1, otherwise zero is returned. The file will be searched for in the device's directory and the program's *media* folder.

FileIsOpen()

Another check you can make before beginning to write or read from a file is to make sure that the file has been successfully opened. This is done using the `FileIsOpen()` statement (see FIG-12.19).

FIG-12.19

`FileIsOpen()`

integer `FileIsOpen` ((`id`))

where:

id is an integer value giving the ID of the file. The will have been assigned by a previous `OpenToWrite()` or `OpenToRead()` statement.

The statement returns 1 if the file has been opened successfully, otherwise zero is returned.

GetFileSize()

We can find out the size of a file in bytes using the `GetFileSize()` statement (see FIG-12.20).

FIG-12.20

`GetFileSize()`

integer `GetFileSize` ((`id`))

where:

id is an integer value giving the ID of the file.

DeleteFile()

If a file is no longer needed it is important that you delete the file to keep the space available on your device to a maximum. File deletion is performed using the `DeleteFile()` statement (see FIG-12.21).

FIG-12.21

`DeleteFile()`

`DeleteFile` ((`file`))

where:

file is a string giving the name of the file to be deleted.

Folder Management

When an app runs it makes use of a folder on your PC's disk or in within your device's memory. This folder is where it finds any resource files needed by your app and where it saves any files created by the app itself. Various AGK commands allow us to discover the path to that folder and the name of the folder itself.

GetWritePath()

The `GetWritePath()` statement allows us to find the path to the folder used by your app. The statement's format is shown in FIG-12.22.

FIG-12.22

GetWritePath()

string `GetWritePath` ()

The path given is an absolute path from the root directory.

GetFolder()

To find out the name of the folder itself, use the `GetFolder()` statement (see FIG-12.23).

FIG-12.23

GetFolder()

string `GetFolder` ()

The statement returns a string containing the name of the folder used. This string will also contain path information if the default folder is no longer being used.

The short program in FIG-12.24 displays the path and folder used by the app.

FIG-12.24

Getting Path and
Folder Details

```
rem *** Get Folder Details ***

rem *** Make the print small enough ***
SetPrintSize(3)
do
    rem *** Display details ***
    Print("Folder used : "+GetWritePath()+ GetFolder())
    Sync()
loop
```

Activity 12.5

Start a new project called *Folders* and implement the code given in FIG-12.22.

Run the project on your PC and then save it.

What is the name of the folder used by the app? Where is the folder located?

Open Windows Explorer and find the folder used by the app.

From the results produced by Activity 12.5 we can see that all files use the folder called *media*. (Don't confuse this with the media folder created within an AGK project when its code is compiled.)

Other platforms will have different path and folder information. In some there will be no specific sub-folder for files and in this case the `GetFolder()` function will

return a blank string.

MakeFolder()

FIG-12.25

You can create a new sub-folder using the `MakeFolder()` statement (see FIG-12.25).

`MakeFolder()`

integer `MakeFolder` (`sfolder`)

where

`sfolder` is a string giving the name of the new sub-folder to be created.

The new folder is created as a sub-folder off the current folder.

Activity 12.6

Modify *Folders* so that it creates a new sub-folder called *MySubFolder* at the start of the program.

Run and save your project. Open Windows Explorer and look inside the executing app's *media* folder and check that the new folder has been created.

SetFolder()

FIG-12.26

To change the folder in which you are working to another folder, use `SetFolder()` (see FIG-12.26).

`SetFolder()`

integer `SetFolder` (`sfolder`)

where

`sfolder` is a string giving the name of the folder

Once you change the folder, your app will look for any resources or data files that need to be loaded in that folder. New files will also be written to the folder.

Activity 12.7

Modify *Folders* by adding a line of code to make *MySubFolder* the new working folder.

After changing the folder, add the lines:

```
OpenToWrite(1,"MyDataFile",0)
WriteString(1,"XXX")
CloseFile(1)
```

Run and save your project.

Open Windows Explorer and check that the new file, *MyDataFile*, has been created within *MySubFolder*.

You can set the current folder back to its original default setting by calling `SetFolder()` with an empty string as the argument.

GetFirstFolder()

To find the name of any other folders which exist as sub-folders to the current folder, begin by using `GetFirstFolder()` (see FIG-12.27).

FIG-12.27

`GetFirstFolder()`

string `GetFirstFolder()`

The function will return the name of the first sub-folder found. If there are no sub-folders, an empty string is returned.

GetNextFolder()

After finding the first sub-folder, subsequent ones should be detected using `GetNextFolder()` (see FIG-12.28).

FIG-12.28

`GetNextFolder()`

string `GetNextFolder()`

Each time the function is called, the next sub-folder's name will be returned. When no more sub-folders remain, the function will return an empty string.

The program in FIG-12.29 demonstrates how the name of all sub-folders within the default start-up folder can be listed.

FIG-12.29

Listing Sub-folders

```
rem *** Finding Sub-folders ***

rem *** Make the print small enough ***
SetPrintSize(3)

rem *** Create some new sub-folders ***
MakeFolder("AAA")
MakeFolder("BBB")
MakeFolder("CCC")

rem *** Get name of first sub-folder ***
name$ = GetFirstFolder()
rem *** List all subfolders ***
while name$ <> ""
    rem *** Print the name ***
    Print(name$)
    rem *** Get the next name ***
    name$ = GetNextFolder()
endwhile
Sync()

do
loop
```

Activity 12.8

Start a new project called *SubFolders* and implement the code given in FIG-12.29.

Test and save your program.

DeleteFolder()

FIG-12.30

An existing folder can be deleted using `DeleteFolder()` (see FIG-12.30).

DeleteFolder()

`DeleteFolder` (`sfolder`)

where

sfolder is a string giving the name of the folder to be deleted. The folder must be a sub-folder of the current directory. There must be no files within the folder to be deleted.

Activity 12.9

Modify *SubFolders* so that after the sub-folders are listed, folder BBB is deleted and the folder names relisted.

Test and save your program.

GetFirstFile()

We can work our way through the names of the files in the current directory in much the same way as we did with the sub-folders. The first filename in the current folder can be found using `GetFirstFile()` (see FIG-12.31).

FIG-12.31

GetFirstFile()

string `GetFirstFile` ()

The function returns the name of the first file in the current folder. If there are no files, an empty string is returned.

GetNextFile()

FIG-12.32

GetNextFile()

The next file name can be obtained using `GetNextFile()` (see FIG-12.32).

string `GetNextFile` ()

The function will return an empty string when no more filenames remain.

File - Zip

Your AGK app can create and access zip files. A zip file is a file which contains a collection of one or more regular files in a compressed format. You can even create a structure of folders and sub-folders within the zip file.

Using a zip file can be a useful way of collecting together all the files used by an app and minimising the storage space those files require.

CreateZip()

Use `CreateZip()` to create a zip file in the app's working folder. The statement's format is shown in FIG-12.33.

FIG-12.33

CreateZip()

Format 1

 A diagram showing the function signature `CreateZip (id , sfile)`. The function name `CreateZip` is in an orange rounded rectangle. It is followed by an opening parenthesis `(`, a green box containing `id`, a comma `,`, a green box containing `sfile`, and a closing parenthesis `)`.
Format 2
 integer 
 A diagram showing the function signature `CreateZip (sfile)`. The function name `CreateZip` is in an orange rounded rectangle. It is followed by an opening parenthesis `(`, a green box containing `sfile`, and a closing parenthesis `)`.

where

id is an integer value giving the ID to be assigned to the ZIP file.

sfile is a string containing the name of the file to be created.

With format 1 you choose the unique ID to be assigned to the file; in format 2, the ID is selected automatically by the app.

AddZipEntry()

The created zip file is initially empty. We now need to add each of the files we want to hold in the zip file. This is done using `AddZipEntry()` which allows us to add a named file to the zip file. The format for this statement is given in FIG-12.34.

FIG-12.34

AddZipEntry()


 A diagram showing the function signature `AddZipEntry (id , sfile , szippath)`. The function name `AddZipEntry` is in an orange rounded rectangle. It is followed by an opening parenthesis `(`, a green box containing `id`, a comma `,`, a green box containing `sfile`, a comma `,`, a green box containing `szippath`, and a closing parenthesis `)`.

where

id is an integer value giving the ID of the existing ZIP file.

sfile is a string giving the name of the file to be added. This string may contain relative path information.

szippath is a string giving the path within the zip file where the new file is to be stored. If the string is empty the file is stored in the root area of the zip. If folder names are given, the folders will be created within the zip file where necessary.

CloseZip()

When all of the necessary files have been added, the zip file needs to be closed. This is done using `CloseZip()` (see FIG-12.35).

FIG-12.35

CloseZip()


 A diagram showing the function signature `CloseZip (id)`. The function name `CloseZip` is in an orange rounded rectangle. It is followed by an opening parenthesis `(`, a green box containing `id`, and a closing parenthesis `)`.

where

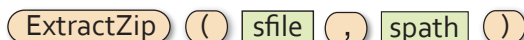
id is an integer value giving the ID of the ZIP file to be closed.

ExtractZip()

The files held within a zip file can be extracted using `ExtractZip()` (see FIG-12.36).

FIG-12.36

ExtractZip()


 A diagram showing the function signature `ExtractZip (sfile , spath)`. The function name `ExtractZip` is in an orange rounded rectangle. It is followed by an opening parenthesis `(`, a green box containing `sfile`, a comma `,`, a green box containing `spath`, and a closing parenthesis `)`.

where

sfile	is a string giving the name of the ZIP file from which other files are to be extracted.
spath	is a string giving details of the folder where the extracted files are to be stored. An empty string will cause the files to be stored in the working folder.

Summary

- Any data that is to exist when an app has finished executing must be saved in a file.
- Files are stored in a folder determined by the device on which the app is being run.
- Use `WriteToFile()` to write to an existing file or to create a new output file.
- New data can be added to existing data within a file or the existing data can be erased.
- Use `WriteInteger()` to write an integer value to a file.
- Use `WriteFloat()` to write a real value to a file.
- Use `WriteString()` to write a standard, null-terminated string to a file.
- Use `WriteLine()` to write a return-terminated string to a file.
- Use `WriteByte()` to write a single byte to a file.
- Use `GetFileExists()` to check that a file exists before attempting to open a file for reading.
- Use `ReadInteger()` to read an integer value from a file.
- Use `ReadFloat()` to read a real value from a file.
- Use `ReadString()` to read a null-terminated string from a file.
- Use `ReadLine()` to read a return-terminated string from a file.
- Use `ReadByte()` to read a single byte from a file.
- Use `FileIsOpen()` to check if a file has been opened successfully.
- Use `GetFileSize()` to determine the number of bytes within a file.
- Use `DeleteFile()` to delete a file that is no longer required.
- Use `GetWritePath()` to discover the path to the main folder used by the app.
- Use `GetFolder()` to find the name of the main folder currently being used by the app.
- Use `MakeFolder()` to create a sub-folder off the current folder.
- Use `SetFolder()` to specify a new main folder.
- Use `GetFirstFolder()` to find the name of the first sub-folder off the current folder.
- Use `GetNextFolder()` to get the names of subsequent sub-folders.

- Use `DeleteFolder()` to delete an empty sub-folder.
- Use `GetFirstFile()` to get the name of the first file within the current folder.
- Use `GetNextFile()` to get the names of subsequent files in the current folder.
- Use `CreateZip()` to create an empty zip file.
- Use `AddZipEntry()` to add a data file to an open zip file.
- Use `CloseZip()` to close a previously opened zip file.
- Use `ExtractZip()` to extract all the data files in a named zip file

Solutions

Activity 12.1

No solution required.

Activity 12.2

Modified code for *UseDataFile*:

```
rem *** Open file for reading ***
myfile = OpenToRead("Test.dat")
rem *** read data from file ***
name$ = ReadString(myfile)
score = ReadInteger(myfile)
rem *** Close the file ***
CloseFile(myfile)
rem *** Display information read ***
Print(name$)
Print(score)
Sync()
do
loop

rem *** set up record stucture ***
type PlayerType
  name as string
  score as integer
endtype
rem *** Create record varibale ***
nol as PlayerType
rem *** Assign values to the fields within the
  records ***
nol.name = "Jane"
nol.score = 613
rem *** Open file ***
myfile = OpenToWrite("Test.dat",0)
rem *** Write record fields to file ***
WriteString(myfile,nol.name)
WriteInteger(myfile,nol.score)
rem *** Close file ***
CloseFile(myfile)
Print("Writing to file completed")
Sync()
do
loop
```

Activity 12.3

Modified code for *LongFile*:

```
rem *** Open file for writing ***
myfile = OpenToWrite("Numbers.dat",0)
rem *** Generate random number ***
no = Random(1,12)
rem *** WHILE no not 12 DO ***
while no <> 12
  rem *** Write number to file ***
  Print(no)
  WriteInteger(myfile,no)
  rem *** Generate another number ***
  no = Random(1,12)
endwhile
Sync()
rem *** Close the file ***
CloseFile(myfile)
do
loop
```

To run on your device, run the app player or viewer then press the compile and broadcast button.

Activity 12.4

Modified code for *LongFile*:

```
rem *** Open file for reading ***
myfile = OpenToRead("Numbers.dat")
rem *** read a value from the file ***
num = ReadInteger(myfile)
rem *** WHILE not EOF DO ***
while fileEOF(myfile) <> 1
  rem *** Display the value read ***
```

```
Print(num)
rem *** Read a value from the file ***
num = ReadInteger(myfile)
endwhile
rem *** Close the file ***
CloseFile(myfile)
Print("Finished")
Sync()
do
loop
```

```
rem *** Open file for writing ***
myfile = OpenToWrite("Numbers.dat",0)
rem *** Generate random number ***
no = Random(1,12)
rem *** WHILE no not 12 DO ***
while no <> 12
  rem *** Write number to file ***
  Print(no)
  WriteInteger(myfile,no)
  rem *** Generate another number ***
  no = Random(1,12)
endwhile
Sync()
rem *** Close the file ***
CloseFile(myfile)
do
loop
```

Activity 12.5

The name of the folder is *media*.

The path, on this occasion was

C:\Users\User\Documents\AGK\Folders\

Activity 12.6

Modified version of *Folders*:

```
rem *** Get Folder Details ***

rem *** Make new sub-folder ***
MakeFolder("MySubFolder")
rem *** Make the print small enough ***
SetPrintSize(2.5)
do
  rem *** Display details ***
  Print("Folder used : " + GetWritePath() +
    GetFolder())
  Sync()
loop
```

The output will not change from the previous version but you should now see the new folder when using Windows Explorer.

Activity 12.7

Modified version of *Folders*:

```
rem *** Get Folder Details ***

rem *** Make new sub-folder ***
MakeFolder("MySubFolder")
rem *** Make this the active folder ***
SetFolder("MySubFolder")
rem *** Write a file to the folder ***
OpenToWrite(1,"MyDataFile",0)
WriteString(1,"XXX")
CloseFile(1)
rem *** Make the print small enough ***
SetPrintSize(2.5)
do
  rem *** Display details ***
  Print("Folder used : " + GetWritePath() +
    GetFolder())
  Sync()
loop
```

The output will now show the result from *GetFolder()* as *media/MySubFolder*.

Using Windows Explorer, we can see that the new file has

been created within *MySubFolder*.

Activity 12.8

No solution required.

Activity 12.9

Modified code for *SubFolders*:

```
rem *** Finding Sub-folders ***

rem *** Make the print small enough ***
SetPrintSize(3)
rem *** Create some new subfolders ***
MakeFolder("AAA")
MakeFolder("BBB")
MakeFolder("CCC")
rem *** Get name of first sub-folder ***
name$ = GetFirstFolder()
rem *** List all the subfolders ***
while name$ <> ""
    rem *** Print the name ***
    Print(name$)
    rem *** Get the next name ***
    name$ = GetNextFolder()
endwhile
Sync()
rem *** Wait 3 seconds ***
Sleep(3000)
rem *** Delete folder BBB ***
DeleteFolder("BBB")
Print ("Deleted folder")
Sync()
Sleep(1000)
rem *** Relist all the folder names ***
name$ = GetFirstFolder()
while name$ <> ""
    Print(name$)
    name$ = GetNextFolder()
endwhile
Sync()
do
loop
```

13

Particles

In this Chapter:

- ☐ Creating Particle Emitters
- ☐ Adjusting Particle Characteristics
- ☐ Applying Forces to Particles
- ☐ Setting Emitter Shape
- ☐ Changing Particle Colours
- ☐ Using An Image to Define Particle Shape
- ☐ Setting Particle Depth
- ☐ Fixing Emitter Screen Position

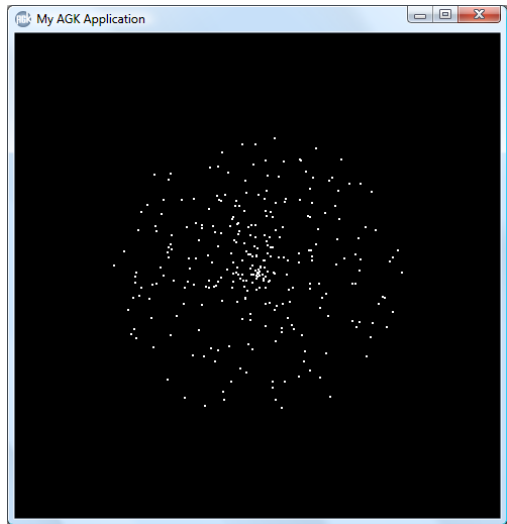
Particles

Introduction

If you've ever seen a sparkler in action on Guy Fawkes night (or whatever day your country uses as an excuse to let you play with fireworks), you'll be able to imagine the effect of the particle instructions in AGK BASIC. Once you've created a particles object, it continues to create a sparkler effect until it is destroyed. A snapshot of the basic effect is shown in FIG-13.1.

FIG-13.1

Particles



The particles shoot out from a central point. This point is known as the **particle emitter**. Like a true firework, the sparks eventually die off and disappear.

There are various characteristics that can be set, such as the quantity, speed, duration and colour of the particles. It will be useful if we think of a particle object as having two components: the emitter and the particles themselves.

Creating Particles

MakeParticles()

To create a stream of particles, we need to execute a `MakeParticles()` statement. This positions the emitter on the screen and assigns an ID to the whole effect. The statement has two possible formats as shown in FIG-13.2.

FIG-13.2

CreateParticles()

Version 1

CreateParticles (id , x , y)

Version 2

integer CreateParticles (x , y)

where:

id is an integer stating the ID to be assigned to the particles object.

x,y are real values giving the position of the emitter within the app window.

In the second version of the statement, the ID is assigned automatically and returned by the statement.

Activity 13.1

Start a new project called *UsingParticles* and modify *main.agc* to contain the following code.

```
rem *** Particles ***  
id = CreateParticles(70,40)  
do  
    Sync()  
loop
```

Modify the coordinates to position the emitter at the centre of the app window.

Test and save the project.

SetParticlesSize()

The size of the particles produced by the emitter can be changed using the `SetParticleSize()` statement (see FIG-13.3).

FIG-13.3

SetParticlesSize()

SetParticlesSize ((id , size)

where:

id is an integer value giving the ID previously assigned to the particles object.

size is a real number giving the size of the particles. The default size is about 0.1.

Activity 13.2

Modify the size of the particles in your *UsingParticles* project setting them to a size of 0.5.

Test and save your project.

SetParticleFrequency()

The number of particles created every second by the emitter can be changed using the `SetParticlesFrequency()` statement (see FIG-13.4).

FIG-13.4

SetParticlesFrequency()

SetParticlesFrequency ((id , ifreq)

where:

id is an integer value giving the ID previously assigned to the particles object.

ifreq is an integer value giving the number of particles to be created every second.

Activity 13.3

Set the particle frequency in your program to 20 and observe what effect this has on the display.

SetParticlesLife()

Particles exist for a fixed amount of time before disappearing. That time can be adjusted using the `SetParticlesLife()` statement (see FIG-13.5).

FIG-13.5

SetParticlesLife()

`SetParticlesLife ((id , itime)`

where:

id is an integer value giving the ID previously assigned to the particles object.

itime is an integer value giving the time (in seconds) that the particles are to survive.

Activity 13.4

Set the particle life to 1 second and test the program. Now change the life to 10 seconds and observe how this affects the result.

SetParticlesMax()

The emitter can be made to stop producing particles after a specified number of particles have been created. This is achieved using the `SetParticlesMax()` statement (see FIG-13.6).

FIG-13.6

SetParticlesMax()

`SetParticlesMax ((id , imax)`

where:

id is an integer value giving the ID previously assigned to the particles object.

imax is an integer value giving the maximum number of particles to be emitted. Use a value of -1 for an infinite number of particles. -1 is the default value.

Activity 13.5

Set the maximum particle count to 100 and test the program.

ResetParticleCount()

If you have used `SetParticlesMax()` to set the number of particles that can be created, the program keeps a count of how many particles have been emitted and stops producing particles when that count reaches the figure specified. You can reset the count to zero using the `ResetParticleCount()` statement. Note that this does not affect the maximum value set, only the count used to see if that maximum value has been reached.

FIG-13.7

ResetParticleCount()

The format for the `ResetParticleCount()` statement is shown in FIG-13.7.

`ResetParticleCount` (`id`)

where:

id is an integer value giving the ID previously assigned to the particles object.

Activity 13.6

In your program's `do...loop` structure, add the following code:

```
if Random(1,800) = 400
    ResetParticleCount(id)
endif
```

Test your modified code.

SetParticlesVelocityRange()**FIG-13.8**

SetParticlesVelocity-Range ()

The speed of the particles leaving the emitter can be set to be within a given range using the `SetParticlesVelocityRange()` statement (see FIG-13.8).

`SetParticlesVelocityRange` (`id` , `min` , `max`)

where:

id is an integer value giving the ID previously assigned to the particles object.

min is a real value giving the minimum speed of a particle leaving the transmitter.

max is a real number giving the maximum speed of a particle leaving the emitter.

Activity 13.7

Immediately before the `do...loop` structure, add a line setting the speed for particles in the range 0.1 to 2.0 and test your code.

SetParticlesAngle()

Particles normally travel away from the emitter in all directions, but it is possible to limit that direction using the `SetParticlesAngle()` statement (see FIG-13.9).

FIG-13.9

SetParticlesAngle()

`SetParticlesAngle` (`id` , `angle`)

where:

id is an integer value giving the ID previously assigned to the particles object.

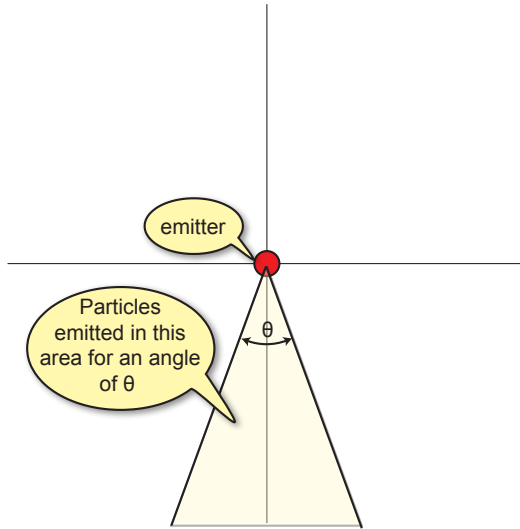
angle is a real value giving the angle over which particles are to be

emitted. The angle is given in degrees.

By default, the angle is measured on either side of a vertical line drawn down from the emitter (see FIG-13.10).

FIG-13.10

How
`SetParticlesAngle()`
Operates



Activity 13.8

Add the statement `SetParticlesAngle(id, 90)` at an appropriate point in your program.

How does this effect the particles produced?

`SetParticlesAngleRad()`

If you wish to specify the particles' angle in radians rather than degrees, you can use the `SetParticlesAngleRad()` statement (see FIG-13.11).

FIG-13.11

`SetParticlesAngleRad()`

`SetParticlesAngleRad` (`id` , `angle`)

where:

id is an integer value giving the ID previously assigned to the particles object.

angle is a real value giving the angle over which particles are to be emitted (in radians).

`SetParticlesDirection()`

The trouble with the `SetParticlesAngle()` statement is that although it gives us control over the angle of spread of the particles, it does not allow us to specify a direction. By default, the spread is always centred around a vertical line starting down from the emitter. To control the direction of that spread we can use the `SetParticlesDirection()` statement (see FIG-13.12).

FIG-13.12

`SetParticlesDirection()`

`SetParticlesDirection` (`id` , `x` , `y`)

where:

id	is an integer value giving the ID previously assigned to the particles object.
x,y	are real values giving the end point of a line starting at the emitter. The emitter is taken as the origin for these coordinates.

Perhaps the simplest way to specify the (x,y) coordinates for this command is to use the `cos()` and `sin()` function. For example, the statement

```
SetParticlesDirection(id,cos(270),sin(270))
```

would emit particles directly upwards.

By using `SetParticlesDirection()` and `SetParticlesAngle()` together you can control the direction and spread of the particles.

Activity 13.9

Add a `SetParticlesDirection()` statement which causes the particles to fly off to the right of the app window.

The `SetParticlesDirection()` statement also has an effect on the speed of the particles. The further the end of the specified line is from the emitter, the faster particles will move (although the particles' speed continues to be affected by any `SetParticlesVelocityRange()` statement that is included in your code). So the statement

```
SetParticlesDirection(id,4*cos(270),4*sin(270))
```

will allow the particles to be up to four times faster than the previous example.

Activity 13.10

Modify your `SetParticlesDirection()` statement so that the end point is ten times greater in distance from the emitter.

The program in FIG-13.13 modifies the particles' direction dynamically while the program is running.

FIG-13.13

Using
`SetParticlesDirection()`

```
rem *** Rotating Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,20)
SetParticlesLife(id,20)
SetParticlesVelocityRange(id,2.0,5.0)
SetParticlesDirection(id,4*cos(0),4*sin(0))
SetParticlesAngle(id,20)
angle = 0
do
    if Random(1,10) = 5
        inc angle,5
        SetParticlesDirection(id,4*cos(angle),4*sin(angle))
    endif
Sync()
loop
```

Activity 13.11

Start a new project called *RotatingParticles* and set *main.agc* to the code shown in FIG-13.13.

Test and save your project.

AddParticlesForce()

Once particles have started moving, they will normally keep going in the same direction and speed until the end of their lifetime. However, by using the `AddParticlesForce()` statement you can adjust that velocity. The format of this statement is shown in FIG-13.14.

FIG-13.14

AddParticlesForce()

AddParticlesForce ((id , start , finish , x , y)

where:

- | | |
|---------------|---|
| id | is an integer value giving the ID previously assigned to the particles object. |
| start | is a real value giving the number of seconds into the particles' life when the effect is to start. |
| finish | is a real value giving the number of seconds into the particles' life when the effect is to finish. |
| x | is a real number giving the force in the x direction that is to be added to the particles' velocity every second during which the force is being applied. |
| y | is a real number giving the force in the y direction that is to be applied to the particles every second. |

For example, the line

```
AddParticlesForce (id,3,5,-15,-5)
```

would, between the 3rd and 5th second of each particle's life, exert a force of -15 in the x direction (towards the left) and -5 in the y direction (upwards).

Activity 13.12

Start a new project called *ParticlesForce*.

In *main.agc*, write code to create a particles emitter at position (10,60). Set particles size to 0.5, frequency to 25 and life to 20. Velocity range should be 2.0 to 5.0 and particle direction should be $4*\cos(0)$, $4*\sin(0)$ with an angle of spread of 20°. Between 3 and 5 seconds, a force of -15,-5 should be applied.

Test and save your project.

Add a second `AddParticlesForce()` statement so that, between 6 and 8 seconds, a force of 15, 5 is applied.

ClearParticlesForces()

If a program makes use of the `AddParticlesForce()` statement to affect the path of the particles, all such forces can be eliminated using the `ClearParticlesForces()` statement which has the format shown in FIG-13.15.

FIG-13.15

ClearParticlesForces()

ClearParticlesForces ((id))

where:

id is an integer value giving the ID previously assigned to the particles object.

As soon as this statement is executed, no new forces will be applied to existing particles, but these will continue on their existing paths. New particles will be unaffected by any previously executed `AddParticlesForce()` statements.

Activity 13.13

In your *ParticlesForce* project, add code so that there is a one chance in 400 of particle forces being eliminated each time the `Sync()` statement is executed.

SetParticlesStartZone()

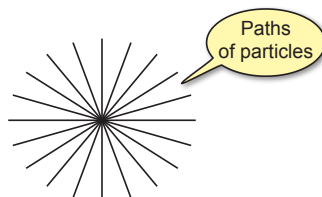
By default, all particles start from a single point on the screen. This point is the position of the emitter. By using the `SetParticlesStartZone()` statement we can create either a line along which particles can start or a rectangular perimeter. The concept is shown in FIG-13.16.

FIG-13.16

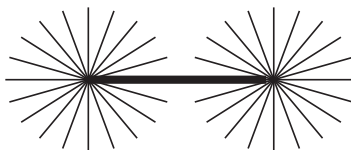
Using Various Start Zone Options

► Remember, although the diagram shows particles being emitted from specific points on the line and rectangle, in fact, they can be emitted from anywhere along the line or rectangle's perimeter.

Normally, all particles originate from a single point...

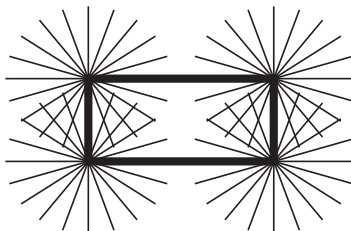


...but they can be made to originate from anywhere along a line...



To simplify the diagram, particles are shown as coming from the two ends of the line only.

...or even anywhere along the perimeter of a rectangle.



To simplify the diagram, particles are shown as coming from the four corners only.

FIG-13.17

The format for the `SetParticleStartZone()` statement is shown in FIG-13.17.

SetParticleStartZone() `SetParticleStartZone` ((`id` , `x1` , `y1` , `x2` , `y2`))

where:

- id** is an integer value giving the ID previously assigned to the particles object.
- x1,y1** are real values giving the coordinates of the top-left corner of the particles' start zone. Coordinates are relative to the emitter.
- x2,y2** are real values giving the coordinates of the bottom-right corner of the particles' start zone. Coordinates are relative to the emitter.

When *y1* and *y2* have the same value, the start zone will be a horizontal line (as shown in the second option of FIG-13.16). When *x1* and *x2* have the same value, the start zone will be a vertical line. For all other options, the start zone will be a rectangle (as shown in the third option of FIG-13.16).

The program in FIG-13.18 demonstrates the use of a horizontal line as the particles' start zone.

FIG-13.18

Using
SetParticleStartZone()

```
rem *** Start zone particles ***
id = CreateParticles(50,50)
SetParticleSize(id,0.5)
SetParticleFrequency(id,200)
SetParticleLife(id,20)
SetParticleVelocityRange(id,0.5,2.0)

rem *** Set start zone ***
SetParticleStartZone(id,-25,0,25,0)
do
    Sync()
loop
```

Activity 13.14

Start a new project, *ParticlesStartZone*, and implement the code given above.

Modify the code so that a rectangular area is used. Change the *y* values in the `SetParticleStartZone()` statement so that the rectangle is 5% above and 5% below the position of the emitter.

Test and save your project.

AddParticleColorKeyFrame()

FIG-13.19

AddParticleColor-
KeyFrame()

If you are bored with white particles, you can add a little colour using the `AddParticleColorKeyFrame()` statement. This statement allows you to specify a new colour for the particles at a specific time in their lifespan. The format of the `AddParticleColorKeyFrame()` statement is shown in FIG-13.19.

`AddParticleColorKeyFrame` ((`id` , `t` , `ir` , `ig` , `ib` , `ia`))

where:

id	is an integer value giving the ID previously assigned to the particles object.
t	is a real number giving the number of seconds into the particles' lifetime when the colour change is to take effect.
ir	is an integer value (0 to 255) giving the red component of the new colour.
ig	is an integer value (0 to 255) giving the green component of the new colour.
ib	is an integer value (0 to 255) giving the blue component of the new colour.
ia	is an integer value (0 to 255) giving the alpha component of the new colour (0: invisible; 255: opaque).

For example, if we wanted the particles to turn yellow after two seconds, we would use the statement:

```
AddParticlesColorKeyFrame(id,2,255,255,0,255)
```

However, when you make use of this statement, you should also explicitly state what colour the particles should be at the start of their life. Hence, when turning the particles yellow after two seconds, we should precede this with the line

```
AddParticlesColorKeyFrame(id,0,255,255,255,255)
```

which will make the particles white at the start of their life.

Activity 13.15

Modify *ParticlesStartZone* so that the particles start white, turn yellow after 2 seconds, red after 5 seconds and blue after 8 seconds. Test and save your project.

SetParticlesColorInterpolation()

If you watched carefully as the particles changed from one colour to the next in your last Activity, you may have noticed that the transition is a subtle one: the particles shift through various shades as they change from their old colour to the new. You can control this transition using the `SetParticlesColorInterpolation()` statement with has the format shown in FIG-13.20.

FIG-13.20

SetParticlesColor-
Interpolation()

SetParticlesColorInterpolation ((id , imode)

where:

id	is an integer value giving the ID previously assigned to the particles object.
imode	is an integer value (0 or 1) giving the transition mode to be used. 0 gives instant transition; 1 (the default) gives smooth transition.

Activity 13.16

Re-run the latest version of *ParticlesStartZone* paying attention to how the particles change colour.

Now add the line

```
SetParticlesColorInterpolation (id,0)
```

to your program and check out how this changes the transition effect.

Resave your project.

ClearParticlesColors()

The colour assignment for new particles can be forced back to the standard white using the `ClearParticlesColors()` statement. However, existing particles will retain their current colour for the remainder of their lifetime.

FIG-13.21

ClearParticlesColors()

`ClearParticlesColors (id)`

where:

id is an integer value giving the ID previously assigned to the particles object.

Activity 13.17

Modify *ParticlesStartZone* so that all new particles revert to white after 10 seconds.

Test and save your project.

SetParticlesImage()

Another option available when dealing with particles, is to replace the simple square, default particle with an image. To do this we need to use the `SetParticlesImage()` statement which has the format shown in FIG-13.22.

FIG-13.22

SetParticlesImage()

`SetParticlesImage (id , imgId)`

where:

id is an integer value giving the ID previously assigned to the particles object.

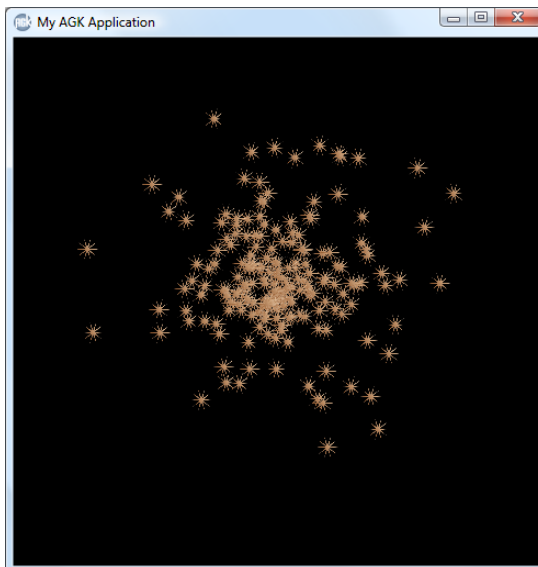
imgId is an integer value giving the ID of the image to be used to create the particles. This image must have been previously loaded using the `LoadImage()` statement.

You will have to make the size of the particles larger to see the image used; perhaps around 3 or 4%.

Output from an image-based set of particles is shown in FIG-13.23.

FIG-13.23

Image-Based
Particles



The program in FIG-13.24 shows how this output was created.

FIG-13.24

Using the
`SetParticlesImage()`
Statement

```
rem *** Image-based Particles ***

rem *** Create particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,4)
SetParticlesFrequency(id,100)
SetParticlesLife(id,2)
SetParticlesVelocityRange(id,0.5,2.0)
rem *** Load image ***
LoadImage(1,"Star.png",0)
rem *** Assign image to particles ***
SetParticlesImage(id,1)
do
    Sync()
loop
```

Activity 13.18

Start a new project called *ImageParticles*. Compile the default code and copy the file *Star.png* from the *AGKDownloads/Chapter13* to the project's *media* folder. Implement the code given in FIG-13.24 and test your program.

Modify the code so that the particles change to yellow after 0.75 seconds and red after 1.25 seconds. Test the new code.

Modify the program again so that the emitter is always positioned at any point pressed on the screen (use `SetParticlesPosition()`). Test your code.

To stop the emitter when the mouse button or your finger is raised, set the particle frequency to zero when `GetPointerState()` returns zero and 100 when `GetPointerState()` returns 1.

Resave your project.

SetParticlesVisible()

You can hide all the particles currently showing on the screen using the `SetParticlesVisible()` statement. You can also make them reappear using the same statement. No particle updating takes place when the particles are invisible, so if the particles are made to reappear, their position and velocity will match those at the time the particles were hidden.

FIG-13.25

The `SetParticlesVisible()` statement has the format shown in FIG-13.25.

`SetParticlesVisible()`

`SetParticlesVisible` (`id` , `invisible`)

where:

id is an integer value giving the ID previously assigned to the particles object.

invisible is an integer value (0 or 1) which determines if the particles are visible (1) or hidden (0).

Activity 13.19

Modify *ImageParticles* so that all particles are invisible when the screen is no longer touched/no mouse button is being pressed and reappear when the screen is next touched/mouse button pressed.

Test and resave your project.

SetParticlesDepth()

If particles are being used in combination with other visual resources such as sprites, we may need to control the “depth” of the particles to make them appear to be “in front” of or “behind” those other elements. This can be achieved using the `SetParticlesDepth()` statement (see FIG-13.26).

FIG-13.26

`SetParticlesDepth()`

`SetParticlesDepth` (`id` , `idepth`)

where:

id is an integer value giving the ID previously assigned to the particles object.

idepth is an integer value giving the depth setting for the emitter and its particles. The default depth for particles and other visual elements is 10. Any value between 0 and 10000 can be used.

FixParticlesToScreen()

Although a topic for a later chapter, the screen display can be zoomed and scrolled. Normally such actions will affect the size and position of an emitter and its particles.

If you want the particles object to be unaffected by these actions, you can make use of the `FixParticlesToScreen()` statement which will fix the particles’ position and size on the screen. The statement can also be used to return the particles to the default option of moving with the scroll. The statement’s format is shown in FIG-13.27.

FIG-13.27

FixParticlesToScreen()

A syntax diagram for the FixParticlesToScreen() function. It consists of the function name 'FixParticlesToScreen' in an orange box, followed by an opening parenthesis '(', the parameter 'id' in a green box, a comma ',' in an orange box, the parameter 'imode' in a green box, and a closing parenthesis ')' in an orange box.

where:

- id** is an integer value giving the ID previously assigned to the particles object.
- imode** is an integer value (0 or 1) used to fix the particles of the screen (1) or to allow them to move and resize with the zooming and scrolling (0).

SetParticlesActive()

You can pause an emitter, freezing its particles in place using `SetParticlesActive()`. Use the same statement to reactivate the flow. The statement has the format shown in FIG-13.28.

FIG-13.28

SetParticlesActive()

A syntax diagram for the SetParticlesActive() function. It consists of the function name 'SetParticlesActive' in an orange box, followed by an opening parenthesis '(', the parameter 'id' in a green box, a comma ',' in an orange box, the parameter 'imode' in a green box, and a closing parenthesis ')' in an orange box.

where

- id** is an integer giving the ID of the emitter.
- imode** is an integer value (0 or 1) which determines the state of the emitter and its particles. (0: inactive, 1: active).

UpdateParticles()

As time passes, we see the particles produced by the emitter move on the screen. If the screen updates 40 times per second, then each frame represents the passing of 0.025 seconds. By using the `UpdateParticles()` statement we can “jump forward in time”. For example, if we were to execute the line

```
UpdateParticles(1,0.5)
```

then the position of the particles from emitter 1 would jump forward 0.5 seconds in time rather than the default 0.025 seconds.

FIG-13.29

The format for the `UpdateParticles()` statement is given in FIG-13.29.

UpdateParticles()

A syntax diagram for the UpdateParticles() function. It consists of the function name 'UpdateParticles' in an orange box, followed by an opening parenthesis '(', the parameter 'id' in a green box, a comma ',' in an orange box, the parameter 'fsecs' in a green box, and a closing parenthesis ')' in an orange box.

where

- id** is an integer value giving the ID of the particle emitter.
- fsecs** is a real number giving the time in seconds that is to be jumped.

Retrieving Particles Data

As we have seen, there are many attributes to a particles object. These include frequency, speed, direction and colour. As well as a set of commands to set these attributes, there is a corresponding set of commands to retrieve their current settings. Those statements are described below.

GetParticlesAngle()

The angle over which particles leave the emitter is set using `SetParticlesAngle()`. The current setting of this angle can be discovered using the `GetParticlesAngle()` statement whose format is given in FIG-13.28.

FIG-13.28

GetParticlesAngle()

float `GetParticlesAngle` ((`id`))

where:

id is an integer value giving the ID previously assigned to the particles object.

The value returned is in degrees.

GetParticlesAngleRad()

When the particles' angle is required in radians rather than degrees, then the `GetParticlesAngleRad()` statement can be used (see FIG-13.29).

FIG-13.29

GetParticlesAngleRad()

float `GetParticlesAngleRad` ((`id`))

where:

id is an integer value giving the ID previously assigned to the particles object.

The value returned is in radians.

GetParticlesDepth()

The current depth setting for an emitter and its particles can be determined using the `GetParticlesDepth()` statement (see FIG-13.30).

FIG-13.30

GetParticlesDepth()

integer `GetParticlesDepth` ((`id`))

where:

id is an integer value giving the ID previously assigned to the particles object.

GetParticlesDirectionX()

When we make use of the `SetParticlesDirection()` statement, we can retrieve the x-coordinate of the end point of the line supplied using `GetParticlesDirectionX()` (see FIG-13.31).

FIG-13.31

GetParticlesDirectionX()

float `GetParticlesDirectionX` ((`id`))

where:

id is an integer value giving the ID previously assigned to the particles object.

GetParticlesDirectionY()

The y-coordinate of the end point of the line can be retrieved using the `GetParticlesDirectionY()` statement (see FIG-13.32).

FIG-13.32

`GetParticlesDirectionY()` float `GetParticlesDirectionY` (`id`)

where:

id is an integer value giving the ID previously assigned to the particles object.

GetParticlesExists()

We can check that an emitter of a specified ID currently exists using the `GetParticlesExists()` statement (see FIG-13.33).

FIG-13.33

`GetParticlesExists()` integer `GetParticlesExists` (`id`)

where:

id is an integer value giving the ID previously assigned to the particles object.

The statement returns 1 if the specified emitter exists, otherwise zero is returned.

GetParticlesFrequency()

We can determine the current rate of particle production from an emitter using the `GetParticlesFrequency()` statement (see FIG-13.34)

FIG-13.34

`GetParticlesFrequency()` float `GetParticlesFrequency` (`id`)

where:

id is an integer value giving the ID previously assigned to the particles object.

GetParticlesLife()

We can discover the current setting for a particle's lifetime using `GetParticlesLife()` (see FIG-13.35).

FIG-13.35

`GetParticlesLife()` float `GetParticlesLife` (`id`)

where:

id is an integer value giving the ID previously assigned to the particles object.

The value returned is given in seconds.

GetParticlesMaxReached()

If we have set a maximum number of particles that are to be emitted, we can find out if that limit has been reached using the `GetParticlesMaxReached()` statement (see FIG-13.36).

FIG-13.36

`GetParticlesMaxReached()` integer `GetParticlesMaxReached` ((id))

where:

id is an integer value giving the ID previously assigned to the particles object.

The statement returns 1 if the set number of particles have been emitted and all of those particles are now dead. All other situations return zero.

GetParticlesSize()

To determine the current setting for particle size, use the `GetParticlesSize()` statement (see FIG-13.37).

FIG-13.37

`GetParticlesSize()` float `GetParticlesSize` ((id))

where:

id is an integer value giving the ID previously assigned to the particles object.

GetParticlesVisible()

A program can check if particles are currently visible using `GetParticlesVisible()` (see FIG-13.38).

FIG-13.38

`GetParticlesVisible()` integer `GetParticlesVisible` ((id))

where:

id is an integer value giving the ID previously assigned to the particles object.

The statement returns 1 if the particles are visible; zero if they are not.

GetParticlesX()

The x-coordinate of the emitter's position can be found using the `GetParticlesX()` statement (see FIG-13.39).

FIG-13.39

`GetParticlesX()` float `GetParticlesX` ((id))

where:

id is an integer value giving the ID previously assigned to the particles object.

GetParticlesY()

The y-coordinate of the emitter position can be found using the `GetParticlesY()` statement (see FIG-13.40).

FIG-13.40

GetParticlesY()

float `GetParticlesY` (`id`)

where:

id is an integer value giving the ID previously assigned to the particles object.

Summary

- A particles object creates a sparkler effect on the screen.
- A particles object consists of an emitter and a set of moving particles which emanate from the emitter.
- By default, particles are white in colour.
- Use `MakeParticles()` to create a particle and position a particles object.
- Each particles object must be assigned a unique ID.
- Use `SetParticlesSize()` to set the size of the particles produced.
- Use `SetParticlesFrequency()` to set the number of particles produced each second.
- Use `SetParticlesLife()` to set the time (in seconds) a particle is to exist.
- Use `SetParticlesMax()` to set the maximum number of particles produced by the emitter. No more particles will be produced after this quantity has been emitted.
- Use `ResetParticleCount()` to reset the count of how many particles have been produced to zero.
- Use `SetParticlesVelocityRange()` to set the minimum and maximum speed of the particles produced. Each particle will be assigned a speed within the range specified.
- Use `SetParticlesAngle()` or `SetParticlesAngleRad()` to set the angle over which the particles will produced. By default this angle is bisected by the positive y-axis.
- Use `SetParticlesDirection()` to set the direction about which the particles' angle is measured.
- Use `AddParticlesForce()` to apply a new force to existing particles after a specified time.
- Use `ClearParticlesForces()` to remove all additional forces applied to particles.
- Use `SetParticlesStartZone()` to set the size and shape of the emitter. This affects where particles first appear.
- Use `AddParticlesColorKeyFrame()` to change the colour of the emitted particles after a given time.

- Use `SetParticlesColorInterpolation()` to specify how particles change from one colour to another (abruptly or through various shade changes).
- Use `ClearParticlesColors()` to return particles to their standard white colour.
- Use `SetParticlesImage()` to use an image to determine the shape of the particles.
- Use `SetParticlesVisible()` to set the particles' visibility.
- Use `SetParticlesDepth()` to set the layer on which particles are placed. The default layer is 10.
- Use `FixParticlesToScreen()` to fix the position of an emitter on the screen, irrespective of any scrolling or zooming effects that are applied.
- Use `GetParticlesAngle()` or `GetParticlesAngleRad()` to retrieve the angle at which particles are being emitted.
- Use `GetParticlesDepth()` to retrieve the depth setting for the particles object.
- Use `GetParticlesDirectionX()` and `GetParticlesDirectionY()` to retrieve the end point used after the `SetParticlesDirection()` statement was executed.
- Use `GetParticlesExist()` to check that a particles object of a specified ID currently exists.
- Use `GetParticlesFrequency()` to discover the current frequency setting for a particles object.
- Use `GetParticlesLife()` to discover the current life setting for a particles object.
- Use `GetParticlesMaxReached()` to discover if the maximum number of particles have been produced.
- Use `GetParticlesSize()` to discover the current size setting for a particles.
- Use `GetParticlesVisible()` to discover the current visibility setting for a particles object.
- Use `GetParticlesX()` and `GetParticlesY()` to discover the position of the particles emitter.

Solutions

Activity 13.1

Modified code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
do
    Sync()
loop
```

Activity 13.2

Modified code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
do
    Sync()
loop
```

Activity 13.3

Modified code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,20)
do
    Sync()
loop
```

Activity 13.4

Final code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,20)
SetParticlesLife(id,10)
do
    Sync()
loop
```

With the particles' life set to one second, the particles do not get far from the emitter before being destroyed. A ten second life gives the particles time to pass the boundary of the app window.

Activity 13.5

Modified code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,20)
SetParticlesLife(id,10)
SetParticlesMax(id,100)
do
    Sync()
loop
```

Particles are no longer produced after exactly 100 particles have appeared.

Activity 13.6

Modified code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,20)
SetParticlesLife(id,10)
SetParticlesMax(id,100)
do
```

```
    if Random(1,800) = 400
        ResetParticleCount(id)
    endif
    Sync()
loop
```

Batches of 100 particles will be produced with random time gaps between each group.

Activity 13.7

Modified code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,20)
SetParticlesLife(id,10)
SetParticlesMax(id,100)
SetParticlesVelocityRange(id,0.1,2.0)
do
    if Random(1,800) = 400
        ResetParticleCount(id)
    endif
    Sync()
loop
```

The speed at which particles leave the emitter will vary; some will be faster than before, some slower.

Activity 13.8

Modified code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,20)
SetParticlesLife(id,10)
SetParticlesMax(id,100)
SetParticlesVelocityRange(id,0.1,2.0)
SetParticlesAngle(id,90)
do
    if Random(1,800) = 400
        ResetParticleCount(id)
    endif
    Sync()
loop
```

All particles moved “down” within a triangular area which extends 45° either side of the vertical.

Activity 13.9

Modified code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,20)
SetParticlesLife(id,10)
SetParticlesMax(id,100)
SetParticlesVelocityRange(id,0.1,2.0)
SetParticlesAngle(id,90)
SetParticlesDirection(id,cos(0),sin(0))
do
    if Random(1,800) = 400
        ResetParticleCount(id)
    endif
    Sync()
loop
```

Notice that, despite the `SetParticlesVelocityRange()` statement, all the particles travel very slowly.

Activity 13.10

Modified code for *UsingParticles*:

```
rem *** Particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,20)
SetParticlesLife(id,10)
SetParticlesMax(id,100)
```

```

SetParticlesVelocityRange(id,0.1,2.0)
SetParticlesAngle(id,90)
SetParticlesDirection(id,10*cos(0),10*sin(0))
do
  if Random(1,800) = 400
    ResetParticleCount(id)
  endif
  Sync()
loop

```

The particles now have a range of speeds similar to that shown before the `SetParticlesDirection()` statement was added.

Activity 13.11

No solution required.

Activity 13.12

Code for *ParticlesForce*:

```

rem *** Particles ***
id = CreateParticles(10,60)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,25)
SetParticlesLife(id,20)
SetParticlesVelocityRange(id,2.0,5.0)
SetParticlesAngle(id,20)
SetParticlesDirection(id,4*cos(0),4*sin(0))
AddParticlesForce(id,3,5,-15,-5)
do
  Sync()
loop

```

Modified code for *ParticlesForce*:

```

rem *** Particles ***
id = CreateParticles(10,60)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,25)
SetParticlesLife(id,20)
SetParticlesVelocityRange(id,2.0,5.0)
SetParticlesAngle(id,20)
SetParticlesDirection(id,4*cos(0),4*sin(0))
AddParticlesForce(id,3,5,-15,-5)
AddParticlesForce(id,6,8,15,5)
do
  Sync()
loop

```

Activity 13.13

Modified code for *ParticlesForce*:

```

rem *** Particles ***
id = CreateParticles(10,60)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,25)
SetParticlesLife(id,20)
SetParticlesVelocityRange(id,2.0,5.0)
SetParticlesAngle(id,20)
SetParticlesDirection(id,4*cos(0),4*sin(0))
AddParticlesForce(id,3,5,-15,-5)
AddParticlesForce(id,6,8,15,5)
do
  if Random(1,400) = 200
    ClearParticlesForces(id)
  endif
  Sync()
loop

```

Activity 13.14

Modified code for *ParticlesStartZone*:

```

rem *** Start zone particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,200)
SetParticlesLife(id,20)
SetParticlesVelocityRange(id,0.5,2.0)
rem *** Set start zone ***
SetParticlesStartZone(id,-25,-5,25,5)
do
  Sync()

```

loop

Activity 13.15

Modified code for *ParticlesStartZone*:

```

rem *** Start zone particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,200)
SetParticlesLife(id,20)
SetParticlesVelocityRange(id,0.5,2.0)
rem *** Set start zone ***
SetParticlesStartZone(id,-25,-5,25,5)
rem *** white particles at start ***
AddParticlesColorKeyFrame(id,0,255,255,255,255)
rem *** yellow particles after 2 seconds ***
AddParticlesColorKeyFrame(id,2,255,255,0,255)
rem *** red particles after 2 seconds ***
AddParticlesColorKeyFrame(id,5,255,0,0,255)
rem *** blue particles after 8 seconds ***
AddParticlesColorKeyFrame(id,8,0,0,255,255)
do
  Sync()
loop

```

Activity 13.16

Modified code for *ParticlesStartZone*:

```

rem *** Start zone particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,200)
SetParticlesLife(id,20)
SetParticlesVelocityRange(id,0.5,2.0)
rem *** Set start zone ***
SetParticlesStartZone(id,-25,-5,25,5)
rem *** white particles at start ***
AddParticlesColorKeyFrame(id,0,255,255,255,255)
rem *** yellow particles after 2 seconds ***
AddParticlesColorKeyFrame(id,2,255,255,0,255)
rem *** red particles after 2 seconds ***
AddParticlesColorKeyFrame(id,5,255,0,0,255)
rem *** blue particles after 8 seconds ***
AddParticlesColorKeyFrame(id,8,0,0,255,255)
rem *** Instant transition ***
SetParticlesColorInterpolation(id,0)
do
  Sync()
loop

```

Activity 13.17

Modified code for *ParticlesStartZone*:

```

rem *** Start zone particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,0.5)
SetParticlesFrequency(id,200)
SetParticlesLife(id,20)
SetParticlesVelocityRange(id,0.5,2.0)
rem *** Set start zone ***
SetParticlesStartZone(id,-25,-5,25,5)
rem *** white particles at start ***
AddParticlesColorKeyFrame(id,0,255,255,255,255)
rem *** yellow particles after 2 seconds ***
AddParticlesColorKeyFrame(id,2,255,255,0,255)
rem *** red particles after 2 seconds ***
AddParticlesColorKeyFrame(id,5,255,0,0,255)
rem *** blue particles after 8 seconds ***
AddParticlesColorKeyFrame(id,8,0,0,255,255)
rem *** Instant transition ***
SetParticlesColorInterpolation(id,0)
rem *** Set timer ***
time = GetSeconds()
do
  rem *** After 10 seconds stop using colour ***
  if GetSeconds()-time = 10
    ClearParticlesColors(id)
  endif
  Sync()
loop

```


Activity 13.18

Modified code for *ImageParticles* (coloured images):

```
rem *** Image-based Particles ***

rem *** Create particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,4)
SetParticlesFrequency(id,100)
SetParticlesLife(id,2)
SetParticlesVelocityRange(id,0.5,2.0)
rem *** Load image ***
LoadImage(1,"Star.png",0)
rem *** Assign image to particles ***
SetParticlesImage(id,1)
rem *** White particles at start ***
AddParticlesColorKeyFrame(id,0,255,255,255,255)
rem *** Yellow particles after 0.75 seconds ***
AddParticlesColorKeyFrame(id,0.75,255,255,0,255)
rem *** Red particles after 1.25 seconds ***
AddParticlesColorKeyFrame(id,1.25,255,0,0,255)
do
    Sync()
loop
```

Modified code for *ImageParticles* (repositioned emitter):

```
rem *** Image-based Particles ***

rem *** Create particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,4)
SetParticlesFrequency(id,100)
SetParticlesLife(id,2)
SetParticlesVelocityRange(id,0.5,2.0)
rem *** Load image ***
LoadImage(1,"Star.png",0)
rem *** Assign image to particles ***
SetParticlesImage(id,1)
rem *** White particles at start ***
AddParticlesColorKeyFrame(id,0,255,255,255,255)
rem *** Yellow particles after 0.75 seconds ***
AddParticlesColorKeyFrame(id,0.75,255,255,0,255)
rem *** Red particles after 1.25 seconds ***
AddParticlesColorKeyFrame(id,1.25,255,0,0,255)
do
    rem *** IF pressed, reposition emitter ***
    if GetPointerState()=1
        SetParticlesPosition(id,GetPointerX(),
            ↵GetPointerY())
    endif
    Sync()
loop
```

Modified code for *ImageParticles* (stop particles when not pressed):

```
rem *** Image-based Particles ***

rem *** Create particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,4)
SetParticlesFrequency(id,100)
SetParticlesLife(id,2)
SetParticlesVelocityRange(id,0.5,2.0)
rem *** Load image ***
LoadImage(1,"Star.png",0)
rem *** Assign image to particles ***
SetParticlesImage(id,1)
rem *** White particles at start ***
AddParticlesColorKeyFrame(id,0,255,255,255,255)
rem *** Yellow particles after 0.75 seconds ***
AddParticlesColorKeyFrame(id,0.75,255,255,0,255)
rem *** Red particles after 1.25 seconds ***
AddParticlesColorKeyFrame(id,1.25,255,0,0,255)
do
    rem *** IF pressed, reposition emitter ***
    if GetPointerState()=1
        SetParticlesFrequency(id,100)
        SetParticlesPosition(id,GetPointerX(),
            ↵GetPointerY())
    else
        SetParticlesFrequency(id,0)
    endif
    Sync()
loop
```

Activity 13.19

Modified code for *ImageParticles*:

```
rem *** Image-based Particles ***

rem *** Create particles ***
id = CreateParticles(50,50)
SetParticlesSize(id,4)
SetParticlesFrequency(id,100)
SetParticlesLife(id,2)
SetParticlesVelocityRange(id,0.5,2.0)
rem *** Load image ***
LoadImage(1,"Star.png",0)
rem *** Assign image to particles ***
SetParticlesImage(id,1)
rem *** White particles at start ***
AddParticlesColorKeyFrame(id,0,255,255,255,255)
rem *** Yellow particles after 0.75 seconds ***
AddParticlesColorKeyFrame(id,0.75,255,255,0,255)
rem *** Red particles after 1.25 seconds ***
AddParticlesColorKeyFrame(id,1.25,255,0,0,255)
do
    rem *** IF pressed, reposition emitter ***
    if GetPointerState()=1
        SetParticlesVisible(id,1)
        SetParticlesPosition(id,GetPointerX(),
            ↵GetPointerY())
    else
        SetParticlesVisible(id,0)
    endif
    Sync()
loop
```


14

Text

In this Chapter:

- ☐ Colouring Text
- ☐ Positioning Text
- ☐ Setting Text Visibility
- ☐ Sizing Text
- ☐ Determining Text Dimensions
- ☐ Adjusting Text Spacing
- ☐ Setting Text Depth
- ☐ Detecting Text Hits
- ☐ Moving and Rotating Individual Characters
- ☐ Colouring Individual Characters
- ☐ Altering Text Display Style
- ☐ Loading a New Default Font
- ☐ Setting the Font for Individual Texts

Introduction

In the first chapters of this book we saw how to create textual output using the `Print()` and `PrintC()` statements. The disadvantage of using these statements for all but the simplest output was obvious; any previously displayed text was lost whenever the screen display was updated (using `Sync()`). The only way to overcome this was to execute the same `Print()` statement after every update.

Later, in Chapter 6, we were introduced to text resources. This allowed us to create text which remained on screen and which could be positioned with accuracy.

In that chapter we covered a few of the basic text commands. In this chapter the remainder of the text commands are covered as well as other techniques such as how to change the font used by the text commands.

Review

We'll start by just listing the text commands covered back in Chapter 6. These were:

<code>CreateText(id, str)</code>	<i>id</i> is the ID to be assigned to the text resource. <i>str</i> is the text to be held in the text resource.
<code>int CreateText(str)</code>	<i>str</i> is the text to be held in the text resource. The ID assigned to the resource is returned.
<code>SetTextColor(id, ir, ig, ib, it)</code>	<i>id</i> is the ID of the text resource <i>ir</i> is the value of the red component (0 to 255). <i>ig</i> is the value of the green component (0 to 255). <i>ib</i> is the value of the blue component (0 to 255). <i>it</i> is the transparency factor (0: invisible; 255:opaque)
<code>SetTextPosition(id, x, y)</code>	<i>id</i> is the ID of the text resource. <i>x, y</i> are real values giving the coordinates for the text.
<code>SetTextSize(id, fsz)</code>	<i>id</i> is the ID of the text resource. <i>fsz</i> is the height of the text (as a percentage or virtual pixels). The width is determined automatically.
<code>SetTextString(id, str)</code>	<i>id</i> is the ID of the text resource. <i>str</i> is the new text to be assigned to the resource. Any previous text is deleted.
<code>SetTextVisible(id, iv)</code>	<i>id</i> is the ID of the text resource. <i>iv</i> determines visibility (1: visible; 0: hidden).
<code>DeleteText(id)</code>	<i>id</i> is the ID of the text resource to be deleted.

SetTextRed(), SetTextGreen(), SetTextBlue() and SetTextAlpha()

As an alternative to setting all the colour and transparency options of text using `SetTextColor()`, you can also set these attributes individually using the `SetTextRed()`, `SetTextGreen()`, `SetTextBlue()` and `SetTextAlpha()` statements. FIG-14.1 to FIG-14.4 give the format for each of these statements.

FIG-14.1

SetTextRed()

`SetTextRed ((id , ired))`

where:

id is an integer value giving the ID of the text resource.

ired is an integer value (0 to 255) giving the intensity of the red. (0: off; 255: full red).

FIG-14.2

SetTextGreen()

`SetTextGreen ((id , igreen))`

where:

id is an integer value giving the ID of the text resource.

igreen is an integer value (0 to 255) giving the intensity of the green. (0: off; 255: full green).

FIG-14.3

SetTextBlue()

`SetTextBlue ((id , iblue))`

where:

id is an integer value giving the ID of the text resource.

iblue is an integer value (0 to 255) giving the intensity of the blue. (0: off; 255: full blue).

FIG-14.4

SetTextAlpha()

`SetTextAlpha ((id , itrans))`

where:

id is an integer value giving the ID of the text resource.

itrans is an integer value (0 to 255) giving the transparency of the text. (0: invisible; 255: opaque).

GetTextRed(), GetTextGreen(), GetTextBlue() and GetTextAlpha()

The colour and transparency values which have been defined for a given text resource can be retrieved using the corresponding `GetTextRed()`, `GetTextGreen()`, `GetTextBlue()` and `GetTextAlpha()` statements (see FIG-14.5 to FIG-14.8).

FIG-14.5

GetTextRed()

integer `GetTextRed ((id))`

FIG-14.6
GetTextGreen()

integer **GetTextGreen** ((id))

FIG-14.7
GetTextBlue()

integer **GetTextBlue** ((id))

FIG-14.8
GetTextAlpha()

integer **GetTextAlpha** ((id))

where:

id is an integer value giving the ID of the text resource whose colour attribute is to be returned.

The value returned will represent the red, green, blue or transparency setting for the text resource depending on which of the four functions is called. All returned values will be in the range 0 to 255.

SetTextX() and SetTextY()

Alternatives to **SetTextPosition()** are **SetTextX()** and **SetTextY()** which allow text to be repositioned horizontally (**SetTextX()**) or vertically (**SetTextY()**). The format for each of these two statements is shown in FIG-14.9 and FIG-14.10.

FIG-14.9

SetTextX()

SetTextX ((id , x))

where:

id is an integer value giving the ID of the text resource.

x is a real value giving the new x-coordinate for the text. This will be a percentage or virtual pixels value.

FIG-14.10

SetTextY()

SetTextY ((id , y))

where:

id is an integer value giving the ID of the text resource.

y is a real value giving the new y-coordinate for the text. This will be a percentage or virtual pixels value.

GetTextX() and GetTextY()

Retrieving the position of a text resource is achieved using the **GetTextX()** and **GetTextY()** statements which return the x and y coordinates respectively. The format for each of these statements is shown in FIG-14.11 and FIG-14.12.

FIG-14.11

GetTextX()

float **GetTextX** ((id))

FIG-14.12

GetTextY()

float **GetTextY** ((id))

where:

id is an integer value giving the ID of the text resource.

The functions return a real number giving the x or y coordinate of the text. This value will represent a percentage or virtual coordinate depending on which system is being used.

GetTextVisible()

To determine if a text resource is visible, use the `GetTextVisible()` statement (see FIG-14.13).

FIG-14.13

GetTextVisible()

integer `GetTextVisible` (`id`)

where:

id is an integer value giving the ID of the text resource.

The function returns 1 if the text is visible; zero if the text is hidden.

GetTextSize()

To discover the current size setting for the text in a text resource, use the `GetTextSize()` statement (see FIG-14.14).

FIG-14.14

GetTextSize()

float `GetTextSize` (`id`)

where:

id is an integer value giving the ID of the text resource.

Activity 14.1

Start a new project called *Text01*, and write a program to determine the default size of a text resource containing the string *Hello again*. Save your project.

GetTextTotalHeight()

A function which performs a similar operation to `GetTextSize()` is `GetTextTotalHeight()` (see FIG-14.15).

FIG-14.15

GetTextTotalHeight()

float `GetTextTotalHeight` (`id`)

where:

id is an integer value giving the ID of the text resource.

This function returns the actual height of the text in the text resource. For a resource containing a single line of text, this will be the same value as `GetTextSize()` returns. However, when an empty string is defined, `GetTextTotalHeight()` will return zero whereas `GetTextSize()` will still return the size setting.

When the text is displayed over several lines, `GetTextTotalHeight()` returns the height from the top of the first line of text to the bottom of the last line.

Activity 14.2

Change the `Print()` statement in *Text01* so that the value returned by `GetTextTotalHeight()` is displayed. Run your program with the original string and then an empty one. Save your project.

GetTextTotalWidth()

When you are placing text within a complex background, it is often necessary to be sure that the text will fit within the space allocated. As well as checking the height of the text we also need to discover the width of the text as a percentage of the screen width or in virtual pixels. This can be achieved using the `GetTextTotalWidth()` statement (see FIG-14.16).

FIG-14.16

`GetTextTotalWidth()`

float `GetTextTotalWidth` (`id`)

where:

id is an integer value giving the ID of the text resource.

Activity 14.3

Modify *Text01* so that both the text height and width are displayed. Run the program using the following text values:

- a) ""
- b) "abc"
- c) "abcdef"

Save your project.

SetTextMaxWidth()

If you need to limit the width of a text item, you can use `SetTextMaxWidth()` (see FIG-14.17).

FIG-14.17

`SetTextMaxWidth()`

`SetTextMaxWidth` (`id` , `fwidth`)

where

id is an integer value giving the ID of the text item.

fwidth is a real number giving the maximum width allowed (in percentage or virtual pixels).

If the text being output exceeds the specified width, it is continued on another line.

This statement must be called before the contents of the text object are set.

The following code

```
CreateText(1, "")
SetTextSize(1, 4)
SetTextMaxWidth(1, 10)
SetTextString(1, "AAABBB")
Sync()
```

produces the output:

```
AAA
BBB
```


SetTextScissor()

You can clip the displayed text so that it does not appear outside a specified rectangular area of the screen. This is done using `SetTextScissor()` (see FIG-14.18).

FIG-14.18

SetTextScissor()

`SetTextScissor ((id , x1 , y1 , x2 , y2))`

where

- id** is an integer value giving the ID of the text item to be affected.
- x1,y1** are real values giving the coordinates of the top-left corner of the rectangular area.
- x2,y2** are real values giving the coordinates of the bottom-right corner of the rectangle.

To understand the effect of this command, compare the output produced from the statement

```
CreateText (1, "ABCDEFGHI"+Chr (10) + "123456789"+Chr (10) + "XXXXXXXXXX  
XXXX")
```

as seen in FIG-14.19

FIG-14.19

Regular Text Output



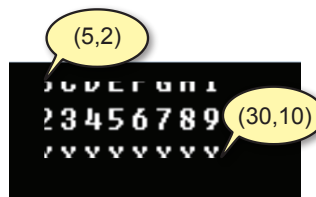
and when the same text object is cut using the line

```
SetTextScissor (1, 5, 2, 30, 10)
```

as shown in FIG-14.20.

FIG-14.20

Clipped Text



GetTextLength()

While `GetTextTotalWidth()` returns the physical size of the string within a text resource, `GetTextLength()` returns the number of characters in the string. This includes any non-printing characters such as *newline*. The format for `GetTextLength()` is shown in FIG-14.21.

FIG-14.21

GetTextLength()

integer `GetTextLength ((id))`

where:

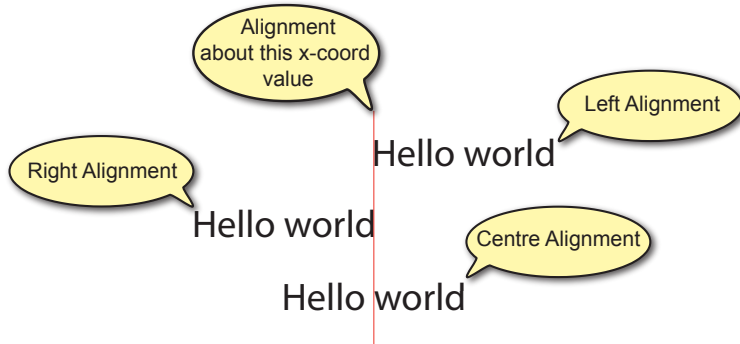
- id** is an integer value giving the ID of the text resource.

SetTextAlignment()

By default, the text position specifies where the top left corner of the text is to be positioned. This is known as left-aligned text. But it is possible to modify this so that the position is used to place the top-right corner of the text (right-aligned text). Finally, the text can be centre-aligned using the position given to place the top-centre of the text. FIG-14.22 shows the effect of each alignment option.

FIG-14.22

Alignment Options



To change from the default left-aligned text, use the `SetTextAlignment()` statement (see FIG-14.23).

FIG-14.23

SetTextAlignment()

```
SetTextAlignment ( ( id , ialign )
```

where:

id is an integer value giving the ID of the text resource.

ialign is an integer value (0, 1 or 2) which gives the alignment to be used. (0: left-alignment, 1: centre-alignment, 2: right-alignment)

Activity 14.4

Modify *Text01* so that the text alignment starts with left-align (the default) then, after a one second delay to right-align and finally, after another one second delay, to centre-align.

Observe the difference in each case then save your project.

SetTextSpacing()

You can create a wider or narrower gap between the individual letters within a piece of text using the `SetTextSpacing()` statement (see FIG-14.24).

FIG-14.24

SetTextSpacing()

```
SetTextSpacing ( ( id , fspace )
```

where:

id is an integer value giving the ID of the text resource.

fspace is a real value giving the gap between the characters in the text. The default value is zero which creates the standard gap. A larger positive value will increase the gap; a negative value will decrease the gap.

Activity 14.5

Modify *Text01* by removing the alignment code added in the previous Activity, then change the gap between the letters of the text to each of the following values:

- a) 1
- b) 2
- c) -1

Observe the difference in each case then save your project

SetTextLineSpacing()

Normally, text resources will contain only a single line of text, but it is possible to have multiple lines by adding the newline character (ASCII code 10) within the text's string with a line such as:

```
CreateText(1, "ABCDEF"+Chr(10)+"GHIJKL")
```

When a multiple-line text resource is being used, you can control the gap between the lines of text using the `SetTextLineSpacing()` statement (see FIG-14.25).

FIG-14.25

SetTextLineSpacing()

`SetTextLineSpacing` ((`id` , `fspace`)

where:

id is an integer value giving the ID of the text resource.

fspace is a real value giving the gap between the lines of the text within the resource. The default value is zero which creates the standard gap. A larger positive value will increase the gap; a negative value will decrease the gap.

Activity 14.6

Change the text string used in *Text01* to contain the text

ABCDEF
GHIJKL
MNOPQR

over three lines.

Now, run the program with line spacing settings of 1, 2 and -1 and observe the difference in each case. Save your project.

SetTextDepth()

Like a sprite, the depth of a text resource can be set. This way you can ensure text always stays on top of any sprite or vice versa. Text depth is set using the statement

`SetTextDepth()` (see FIG-14.26).

FIG-14.26

SetTextDepth()

`SetTextDepth` ((`id` , `idepth`)

where:

- id** is an integer value giving the ID of the text resource.
- idepth** is an integer value (0 to 10,000) giving the depth setting for the text resource. 0 is the front-most layer and will show over any other visual resource.

GetTextDepth()

To discover the current depth setting for a text resource, use `GetTextDepth()` (see FIG-14.27).

FIG-14.27

GetTextDepth()

integer `GetTextDepth` (`id`)

where:

- id** is an integer value giving the ID of the text resource.

GetTextHitTest()

Like sprites, text resources can detect if a given set of coordinates are inside the display area of the text. This is done using the `GetTextHitTest()` statement (see FIG-14.28).

FIG-14.28

GetTextHitTest()

integer `GetTextHitTest` (`id` , `x` , `y`)

where:

- id** is an integer value giving the ID of the text resource.
- x,y** are a pair of real values giving the coordinates to be checked.

If the coordinates given are inside the area of the text resources, this function returns 1, otherwise zero is returned.

Activity 14.7

Start a new project called *MenuSelection*. Get the program to display three text resources containing the strings:

New
Open
Save

The program should display the text contained in any text resource that is pressed. Test and save your project.

GetTextExists()

Any resource that can be deleted should really be checked to make sure it exists before your code attempts to access it. Failure to do this can cause a runtime error.

FIG-14.29

GetTextExists()

integer `GetTextExists` (`id`)

where:

id is an integer value giving the ID of the text resource whose existence is to be checked.

The function returns 1 if the ID specified is assigned to a current text resource, otherwise zero is returned.

Text Character Statements

As well as the set of commands we have looked at so far, all of which affect the characteristics of the whole text resources, another set of commands are available which deal with the individual characters within the text. These statements are covered in this section.

SetTextCharPosition()

We can reposition individual characters within a text resource using the `SetTextCharPosition()` statement (see FIG-14.30).

FIG-14.30

SetTextCharPosition()

`SetTextCharPosition ((id , ichrsub , x , y)`

where:

id is an integer value giving the ID of the text resource.

ichrsub is an integer value giving the position of the character within the text resource's string. The first character is at position zero. If the subscript given is invalid, the statement is not executed.

x,y are a pair of real values giving the new position for the character. The new position is measured relative to the top-left of the first character in the text. All other characters will be unaffected.

The code in FIG-14.31 moves the third letter of a string to a new position after a 3 second delay.

FIG-14.31

Moving a Character

```
rem ** Move a character **

rem *** Create and place text ***
CreateText(1,"ABCDEF")
SetTextPosition(1,50,50)
Sync()
rem *** Wait 3 seconds ***
Sleep(3000)
rem *** Move character ***
SetTextCharPosition(1,2,2,-5)
Sync()
do
loop
```

Activity 14.8

Start a new project called *MoveText* and implement the code given in FIG-14.31.

SetTextCharX() and SetTextCharY()

If you need to move a character one direction only, you can use the `SetTextCharX()` or `SetTextCharY()` statements (see FIG-14.32 and FIG-14.33).

FIG-14.32

SetTextCharX()

`SetTextCharX ((id , ichrsub , x)`

where:

- id** is an integer value giving the ID of the text resource.
- ichrsub** is an integer value giving the position of the character within the text resource's string. The first character is at position zero. If the subscript given is invalid, the statement is not executed.
- x** is a real number giving the character's new position along the x-axis measured from the position of the first character in the existing string.

FIG-14.33

SetTextCharY()

`SetTextCharY ((id , ichrsub , y)`

where:

- id** is an integer value giving the ID of the text resource.
- ichrsub** is an integer value giving the position of the character within the text resource's string. The first character is at position zero. If the subscript given is invalid, the statement is not executed.
- y** is a real number giving the character's new position along the y-axis measured from the position of the first character in the existing string.

The program in FIG-14.34 displays a text containing six letters. Each time the pointer moves over them, a new letter falls to the bottom of the screen.

FIG-14.34

Falling Characters

```
rem ** Move a character ***

rem *** Create and place text ***
CreateText(1,"ABCDEF")
SetTextPosition(1,50,50)
Sync()
rem *** Start a character zero ***
char = 0
do
    rem *** If pointer over text drop first letter remaining ***
    if GetTextHitTest(1,GetPointerX(),GetPointerY()) and char < 6
        for c = 1 to 40
            SetTextCharY(1,char,c)
            Sync()
        next c
        rem *** Move to next letter ***
        inc char
    endif
loop
```

Activity 14.9

Start a new project called *FallingText* and implement the code given in FIG-14.34. Test and save your project.

GetTextCharX() and GetTextCharY()

You can find the coordinates of an individual character within the text using the `GetTextCharX()` and `GetTextCharY()` statements (see FIG-14.35 and FIG-14.36).

FIG-14.35

GetTextCharX()

float `GetTextCharX` ((`id` , `ichrsub`)

where:

- id** is an integer value giving the ID of the text resource.
- ichrsub** is an integer value giving the position of the character within the text resource's string. The first character is at position zero. If the subscript given is invalid, the statement is not executed.

The function returns the x-coordinate of the character relative to the position of the top-left of the parent text.

FIG-14.36

GetTextCharY()

float `GetTextCharY` ((`id` , `ichrsub`)

where:

- id** is an integer value giving the ID of the text resource.
- ichrsub** is an integer value giving the position of the character within the text resource's string. The first character is at position zero. If the subscript given is invalid, the statement is not executed.

The function returns the y-coordinate of the character relative to the position of the top-left of the parent text.

To calculate the absolute position of the fourth character in text resource 1, you can use the lines:

```
x = GetTextX(1) + GetTextCharX(1,3)
y = GetTextY(1) + GetTextCharY(1,3)
```

SetTextCharAngle() and SetTextCharAngleRad()

Characters within a text resource can also be rotated. This is done using either `SetTextCharAngle()` and specifying the angle in degrees or with `SetTextCharAngleRad()` giving the angle in radians. The format of the two functions are shown in FIG-14.37 and FIG-14.38.

FIG-14.37

SetTextCharAngle()

`SetTextCharAngle` ((`id` , `ichrsub` , `angle`)

where:

- id** is an integer value giving the ID of the text resource.

ichrsub is an integer value giving the position of the character within the text resource's string. The first character is at position zero.

angle is a real value giving the angle (in degrees) to which the character is to be rotated.

FIG-14.38

SetTextCharAngleRad()

SetTextCharAngleRad ((id , ichrsub , angle)

where:

id is an integer value giving the ID of the text resource.

ichrsub is an integer value giving the position of the character within the text resource's string. The first character is at position zero.

angle is a real value giving the angle (in radians) to which the character is to be rotated.

The program in FIG-14.39 is a variation on the previous program. This time the letters rotate by a random amount as they fall to the bottom of the screen.

FIG-14.39

Rotating Characters

```
rem ** Move a character ***
rem *** Create and place text ***
CreateText(1,"ABCDEF")
SetTextPosition(1,50,50)
Sync()
rem *** Start a character zero ***
char = 0
do
  rem *** If pointer over text drop first letter remaining ***
  if GetTextHitTest(1,GetPointerX(),GetPointerY()) and char < 6
    rem *** select a random angle ***
    angle = Random(1,360)
    rem *** Calculate 40th of angle ***
    anglestep# = angle / 40.0
    rem *** Current rotation is zero ***
    currentangle# = 0
    rem *** FOR 40 times DO ***
    for c = 1 to 40
      rem *** Move the down 1% ***
      SetTextCharY(1,char,c)
      rem *** Increase angle ***
      currentangle# = currentangle# + anglestep#
      SetTextCharAngle(1,char,currentangle#)
      Sync()
    next c
    rem *** Move to next letter ***
    inc char
  endif
loop
```

Activity 14.10

Modify *FallingText* to match the code given in FIG-14.39. Change the code again so that all characters fall at the same time.

(HINT: This is a more complex change than it first appears. You should use an array for the angle step sizes.) What problems are there with the final result? Save your project.

GetTextCharAngle() and GetTextCharAngleRad()

You can retrieve the current angle setting for a character using either `GetTextCharAngle()` which returns the angle in degrees or `GetTextCharAngleRad()` which returns the angle in radians. The format for each of these two statements is shown in FIG-14.40 and FIG-14.41.

FIG-14.40

`GetTextCharAngle()`

float `GetTextCharAngle` ((`id` , `ichrsub`)

where:

id is an integer value giving the ID of the text resource.

ichrsub is an integer value giving the position of the character within the text resource's string. The first character is at position zero.

FIG-14.41

`GetTextCharAngleRad()`

float `GetTextCharAngleRad` ((`id` , `ichrsub`)

where:

id is an integer value giving the ID of the text resource.

ichrsub is an integer value giving the position of the character within the text resource's string. The first character is at position zero.

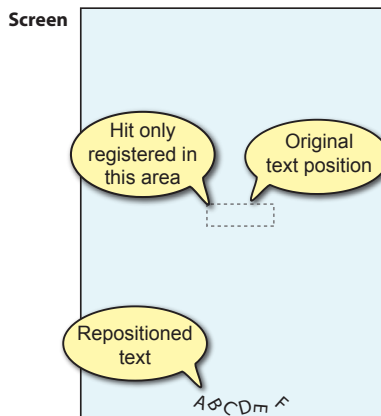
Repositioned Characters and Hit Detection

As we have already seen, it is possible to determine if a point is within a text resource on the screen using the `GetTextHitTest()` statement.

However, this statement uses the original position of the text string to determine if a point (x,y) is within the text string area. It does not take into account any repositioned characters (see FIG-14.42).

FIG-14.42

Detecting a Text Hit



Activity 14.11

Re-run *FallingText*. After all the characters have fallen to the bottom of the screen, try moving the pointer over these characters, then try moving the pointer over the original position in the middle of the screen. What happens in both cases?

SetTextCharColor()

FIG-14.43

The colour of an individual character can be set using `SetTextCharColor()`. This statement has the format shown in FIG-14.43.

SetTextCharColor()

`SetTextCharColor ((id , ichrsub , ired , igreen , iblue , itrans)`

where:

- | | |
|----------------|--|
| id | is an integer value specifying the ID of the text resource whose colour is to be set. |
| ichrsub | is an integer value giving the position of the character within the text resource's string. The first character is at position zero. |
| ired | is an integer value specifying the intensity of the red component of the colour. Range 0 to 255. |
| igreen | is an integer value specifying the intensity of the green component of the colour. Range 0 to 255. |
| iblue | is an integer value specifying the intensity of the blue component of the colour. Range 0 to 255. |
| itrans | is an integer value specifying the transparency of the text. Range 0 (invisible) to 255 (fully opaque). |

SetTextCharColorAlpha(), SetTextCharColorRed(), SetTextCharColorGreen() and SetTextCharColorBlue()

While `SetTextCharColor()` is fine if you want to change all or most of the color attributes of a character, when only one value needs to be changed, then you can use `SetTextCharColorAlpha()`, `SetTextCharColorRed()`, `SetTextCharColorGreen()` or `SetTextCharColorBlue()` as appropriate. The format for each of these statements is shown in FIG-14.44 to FIG-14.47.

FIG-14.44

SetTextCharColorAlpha()

`SetTextCharColorAlpha ((id , ichrsub , itrans)`

where:

- | | |
|----------------|--|
| id | is an integer value giving the ID of the text resource. |
| ichrsub | is an integer value giving the position of the character within the text resource's string. The first character is at position zero. |
| itrans | is an integer value (0 to 255) giving the transparency of the text. (0: invisible; 255: opaque) . |

FIG-14.45

SetTextCharColorRed()

`SetTextCharColorRed ((id , ichrsub , ired)`

where:

- | | |
|-----------|---|
| id | is an integer value giving the ID of the text resource. |
|-----------|---|

ichrsb is an integer value giving the position of the character within the text resource's string. The first character is at position zero.

ired is an integer value (0 to 255) giving the intensity of the red. (0: off; 255: full red).

FIG-14.46

SetTextCharColorGreen() **SetTextCharColorGreen** ((**id** , **ichrsb** , **igreen**))

where:

id is an integer value giving the ID of the text resource.

ichrsb is an integer value giving the position of the character within the text resource's string. The first character is at position zero.

igreen is an integer value (0 to 255) giving the intensity of the green. (0: off; 255: full green).

FIG-14.47

SetTextCharColorBlue() **SetTextCharColorBlue** ((**id** , **ichrsb** , **ibblue**))

where:

id is an integer value giving the ID of the text resource.

ichrsb is an integer value giving the position of the character within the text resource's string. The first character is at position zero.

ibblue is an integer value (0 to 255) giving the intensity of the blue. (0: off; 255: full blue).

GetTextCharColorAlpha(), GetTextCharColorRed(), GetTextCharColorGreen() and GetTextCharColorBlue()

To retrieve the colour and transparency settings for a character, you can use the statements `GetTextCharColorAlpha()`, `GetTextCharColorRed()`, `GetTextCharColorGreen()` and `GetTextCharColorBlue()`. The format for each of these statements is shown in FIG-14.48 to FIG-14.51.

FIG-14.48

GetTextCharColorAlpha() integer **GetTextCharColorAlpha** ((**id** , **ichrsb**))

FIG-14.49

GetTextCharColorRed() integer **GetTextCharColorRed** ((**id** , **ichrsb**))

FIG-14.50

GetTextCharColorGreen() integer **GetTextCharColorGreen** ((**id** , **ichrsb**))

FIG-14.51

GetTextCharColorBlue() integer **GetTextCharColorBlue** ((**id** , **ichrsb**))

where:

id is an integer value giving the ID of the text resource.

ichrsb is an integer value giving the position of the character within

the text resource's string. The first character is at position zero.

The value returned will represent the transparency, red, green, or blue setting for the character depending on which of the four functions is called. All returned values will be in the range 0 to 255.

The program in FIG-14.52 cycles through the characters in a string changing each to yellow and then back to white.

FIG-14.52

Colouring Characters

```
rem *** Colour characters ***
rem *** Create and size text ***
CreateText(1,"oooooooooo")
SetTextSize(1,10)
do
    rem *** FOR each character DO ***
    for c = 0 to 9
        rem *** Change colour to yellow ***
        SetTextCharColor(1,c,255,255,0,255)
        Sync()
        Sleep(200)
        rem *** Change back to white ***
        SetTextCharColorBlue(1,c,255)
        Sync()
    next c
loop
```

Activity 14.12

Create a new project called *ColourfulCharacters* and implement the code given in FIG-14.52.

Modify your program so that each letter takes on a random colour.

Save your project.

SetTextDefaultMinFilter() and SetTextDefaultMagFilter()

The text displayed when you use a `Print()` statement or a text resource is actually taken from an image. When you create text which is smaller or larger than the actual size of the character's image, then AGK scales that image. The scaling process used when the characters are smaller than the image size can be set using `SetTextDefaultMinFilter()`. When the characters are larger than the image size, the scaling process used can be set using `SetTextDefaultMagFilter()`.

The two scaling options available will create either a sharp but blocky result, or a more blurred but smoother finish (see FIG-14.53).

FIG-14.53

Character Display Styles



FIG-14.54

The format for each of the two statements is shown in FIG-14.54 and FIG-14.55.

SetTextDefaultMinFilter()

SetTextDefaultMinFilter ((ioption))

FIG-14.55

SetTextDefaultMagFilter()

SetTextDefaultMagFilter ((ioption))

where:

ioption

is an integer value (0 or 1) giving the scaling process to be used on characters which are not the same size as the original character image. 0: sharp/blocky, 1: blurred/smooth. The default option is 1.

Activity 14.13

Run *ColourfulCharacters* which you created in Activity 14.11 and observe the nature of the characters displayed.

Add the line

```
SetTextDefaultMagFilter(0)
```

at the start of the program and then check what effect this has on the appearance of the characters.

SetTextDefaultFontImage()

The image file used for the characters displayed by the `Print()` statement and text resources can be replaced by your own image containing a new font.

The first choice you have to make is whether to use a monospaced font (where every character is exactly the same width) or a proportional font (where the width of characters vary).

For a monospaced font your image must conform to the following rules:

- Characters must be in 6 rows of 16 characters.
- The characters within a row must be evenly spaced.
- Each row must be evenly spaced.
- The characters must be in ASCII code order, starting with the *space* character (ASCII 32) and ending with the *del* character (ASCII 127).
- The characters must be white on a transparent background.
- The image should be in PNG format.
- The image width must be exactly divisible by 16 and the height by 6.

The image must, like any other, be stored in a project's *media* folder.

The `SetTextDefaultFontImage()` allows us to specify the image containing the font to be used for all subsequent text displays. The format for the statement is given in FIG-14.56.

FIG-14.56

SetTextDefaultFontImage()

SetTextDefaultFontImage ((imgId))

where:

imgId is an integer value giving the ID of the image containing the new font.

This command must be executed before any text resource is created or **Print** statement output produced.

A typical mono-spaced image is shown in FIG-14.57. Note that the background colour is shown in grey only to make the characters visible here - in fact, the background must be transparent. Rectangular blocks are used to replace characters not available in this font.

FIG-14.57

Font Image File



The program in FIG-14.58 uses this font image to demonstrate the result achieved by creating you own font.

FIG-14.58

Using a New Default Font

```
rem *** Load a new font ***
LoadImage(1,"OCRFont.png")
rem *** Create and size text ***
SetTextDefaultFontImage(1)
CreateText(1,"ABCDEFGHJKLMNOPQRSTUVWXYZ"+chr(10)+"abcdefghijklmn
opqrstuvwxyz"+Chr(10)+"AbCdEfGhIjKl")
SetTextSize(1,10)
Sync()
do
loop
```

Activity 14.14

Start a new project called *NewFont*. Copy the file *OCRFont.png* from *AGKDownloads/Chapter14* to the project's *media* folder.

Implement the code given in FIG-14.58 and observe the new font.

When you want to create a proportional font with characters occupying various

widths (e.g. *W* will be wider than *i*) things become more complicated.

You need to create an **atlas texture image file** containing the characters from the *space* (ASCII 32) to the *del* (ASCII 127) character and, to accompany this, you also need a **subimage text file** giving the details of the position and size of each character.

Atlas texture files and the subimage text file formats are explained in Chapter 16.

SetTextDefaultExtendedFontImage()

Regular printable ASCII characters have codes between 32 and 127, but there is also a set of extended characters which use the codes 128 to 255. You can set these up in a text object with statements such as

```
SetTextString(1,Chr(170)+Chr(190))
```

As with the codes 32 to 127, AGK contains a single image with a shape for each of these extended characters.

You can also have your own set of characters for code 128 to 255 by using `SetTextDefaultExtendedFontImage()`. The format for this statement is shown in FIG-14.59.

FIG-14.59

SetTextDefaultExtended
FontImage()

`SetTextDefaultExtendedFontImage ((imgId))`

where

imgId is an integer value giving the ID of the image containing the character images.

The image you use has the same general requirements as those listed earlier for use with `SetTextDefaultFont()`. The only difference being that the extended font image requires 8 rows of 16 characters (since this time there are 128 characters rather than 96).

SetTextFontImage()

If you want to change the font for just one text resource rather than them all, then you can use `SetTextFontImage()`. This statement has the format shown in FIG-14.60.

FIG-14.60

SetTextFontImage()

`SetTextFontImage ((id , imgId))`

where:

id is an integer value specifying the ID of the text resource whose font is to be set.

imgId is an integer value giving the ID of the image containing the new font. The image structure requirements are identical to those described for `SetTextDefaultFont()`.

Summary

- A text resource can be used in place of output produced by a `Print()` statement.

- Text strings do not have to be re-output after every `Sync()` statement, unlike the characters produced by a `Print()` statement.
- Text can be sized, coloured, made transparent and positioned.
- Use `SetTextRed()`, `SetTextGreen()` and `SetTextBlue()` to set the individual colour components of a text object.
- Use `GetTextRed()`, `GetTextGreen()` and `GetTextBlue()` to discover the individual colour component settings of a text object.
- Use `SetTextAlpha()` to set the transparency of a text object.
- Use `GetTextAlpha()` to discover the transparency setting of a text object.
- Use `SetTextX()` and `SetTextY()` to set the individual x and y coordinates of a text object.
- Use `GetTextX()` and `GetTextY()` to discover the individual x and y coordinates of a text object.
- Use `GetTextVisible()` to discover the current visibility setting of a text object.
- Use `GetTextSize()` to discover the current size setting of a text object.
- Use `GetTextTotalHeight()` to discover the overall height of a text object. This will differ from the value returned by `GetTextSize()` if the text extends over several lines or contains zero characters.
- Use `GetTextTotalWidth()` to discover the overall width of a text object.
- Use `SetTextMaxWidth()` to set a maximum width for a text object. If the text exceeds this length it will wrap onto a new line.
- Use `SetTextScissor()` to crop the area of the screen in which a text object is visible.
- Use `GetTextLength()` to discover the number of characters in a text object.
- Use `SetTextAlignment()` to set the alignment of a text object.
- Use `SetTextSpacing()` to set the spacing between the individual characters within a text object.
- Use `SetTextLineSpacing()` to set the spacing between lines of a multi-line text object.
- Use `SetTextDepth()` to set the layer on which a text object is to be placed. The default layer is 10.
- Use `GetTextDepth()` to discover the layer setting of a text object.
- Use `GetTextHitTest()` to discover if a given position is within the text object's space.
- Use `GetTextExists()` to check that a text object of a given ID currently exists.
- Individual characters within a text string can be coloured, made transparent, rotated and positioned.
- Use `SetTextCharPosition()` to set the position of an individual character within a text object.

- Use `SetTextCharX()` and `SetTextCharY()` to set the x and y coordinates of an individual character within a text object.
- Use `GetTextCharX()` and `GetTextCharY()` to discover the x and y coordinates of an individual character within a text object.
- Use `SetTextCharAngle()` or `SetTextCharAngleRad()` to set the angle of an individual character within a text object.
- Use `GetTextCharAngle()` or `GetTextCharAngleRad()` to discover the angle of an individual character within a text object.
- The new position of a moved character is not taken into account when determining if a given point is within the area of a text object.
- Use `SetTextCharColor()` to set the colour of an individual character within a text object.
- Set individual character colour attributes using `SetTextCharRed()`, `SetTextCharGreen()`, `SetTextCharBlue()` and `SetTextCharAlpha()`.
- Use `GetTextCharRed()`, `GetTextCharGreen()`, `GetTextCharBlue()` and `GetTextCharAlpha()` to retrieve the attributes of individual characters within a text object.
- The characters displayed by a text object are obtained from an image.
- When characters are displayed at a size other than that within the image from which they are taken, those characters are scaled.
- Scaling uses one of two options: sharp but blocky, or smooth but blurred (the default).
- Use `SetTextDefaultMinFilter()` to select which scaling option is used for characters smaller than the original size.
- Use `SetTextDefaultMagFilter()` to select which scaling option is used for characters larger than the original size.
- The font used by all text resources can be changed by loading a new image file containing a new font.
- New font image files can be monospaced or proportional.
- Monospace font images must be 16 characters by 6 rows covering the characters *space to del.*
- Proportional font images must be accompanied by a subimage text file giving the position and dimensions of each character.
- The subimage text file accompanying a proportional font image must give filenames based on each character's ASCII code value (e.g. *65.png* for the upper case *A*).
- Use `SetTextDefaultFontImage()` to set the image used when displaying all text characters in the ASCII range 32 to 127.
- Use `SetTextDefaultExtendedFontImage()` to set the image used when displaying characters in the ASCII range 128 to 255.
- The image used for characters 128 to 255 should be 16 characters by 8 rows for a monospaced font.

- Use `SetTextFontImage()` to change the image used for a specific text object (all other text objects will use the default image).

Solutions

Activity 14.1

Code for *Text01*:

```
CreateText(1,"Hello again")
SetTextPosition(1,50,50)
Print(GetTextSize(1))
Sync()
do
loop
```

It is necessary to move the text resource away from its default position at the top left of the screen since the `Print()` statement also writes to that position.

The default size is 4.0.

Activity 14.2

Modified code for *Text01*:

```
CreateText(1,"Hello again")
SetTextPosition(1,50,50)
Print(GetTextTotalHeight(1))
Sync()
do
loop
```

To change the text, modify the first line to read:

```
CreateText(1,"")
```

Notice that the height given for "Hello again" matches that returned by `GetTextSize()`, but in the case of the empty string, `GetTextTotalHeight()` returns zero since no actual text is output.

Activity 14.3

Modified code for *Text01*:

```
CreateText(1,"")
SetTextPosition(1,50,50)
Print(GetTextTotalHeight(1))
Print(GetTextTotalWidth(1))
Sync()
do
loop
```

A blank string returns a length of zero.

"abc" returns 7.5.

"abcdef" returns 15.0.

Since the text width is given as a percentage of the screen's width, then the text width values will change if the screen width changes.

Activity 14.4

Modified code for *Text01*:

```
CreateText(1,"abcdef")
rem *** Left-alignment ***
SetTextAlignment(1,0)
SetTextPosition(1,50,50)
Sync()
Sleep(1000)
rem *** right-alignment ***
SetTextAlignment(1,2)
Sync()
Sleep(1000)
rem *** centre-alignment ***
SetTextAlignment(1,1)
Sync()
Print(GetTextTotalHeight(1))
Print(GetTextTotalWidth(1))
Sync()
do
loop
```

Activity 14.5

Modified code for *Text01*:

```
CreateText(1,"abcdef")
SetTextPosition(1,50,50)
SetTextSpacing(1,1)
Print(GetTextTotalHeight(1))

Print(GetTextTotalWidth(1))
Sync()
do
loop
```

The `SetTextSpacing()` command is changed to

```
SetTextSpacing(1,2)
```

and

```
SetTextSpacing(1,-1)
```

for subsequent runs.

Not only is there an obvious visual effect, but the value returned by the `GetTextTotalWidth()` statement will also change. The values returned by this are:

20.0, 25.0 and 10

Activity 14.6

Modified code for *Text01*:

```
CreateText(1,"abcdef"+chr(10)+"ghijkl"+chr(10)+
% "mnopqr")
SetTextPosition(1,50,50)
SetTextSpacing(1,-1)
SetTextLineSpacing(1,1)
Print(GetTextTotalHeight(1))
Print(GetTextTotalWidth(1))
Sync()
do
loop
```

The `SetTextLineSpacing()` command is changed to

```
SetTextLineSpacing(1,2)
```

and

```
SetTextLineSpacing(1,-1)
```

for subsequent runs.

The output produced by `GetTextTotalHeight()` will be:

14.0, 16.0 and 10.0

Activity 14.7

Code for *MenuSelection*:

```
rem *** Menu selection ***

rem *** Set up strings ***
dim menu[3] as string = ["","New","Open","Save"]
rem *** Set up Text resources ***
for c = 1 to 3
  CreateText(c,menu[c])
  SetTextPosition(c, 40,20+c*5)
next c
do
  rem *** Check if pressed ***
  for c = 1 to 3
    if GetTextHitTest(c, GetPointerX(),
      GetPointerY()) = 1 and GetPointerState() = 1
      Print(menu[c])
    endif
  next c
  Sync()
loop
```

The most difficult part is to know what value is in each text resource since there is no `GetTextString()` statement to allow us to discover that value.

To solve this problem, the strings to be placed within the text resources are held in an array, *menu*. The content of *menu[1]* is stored in text ID 1, *menu[2]* in text 2, etc. When we have the ID of a resource we can find the string it contains in the corresponding element of *menu*.

Activity 14.8

No solution required.

Activity 14.9

No solution required.

Activity 14.10

Modified code for *FallingText*:

```
rem ** Move all characters **
dim anglesteps#[6]
rem *** Create and place text ***
CreateText(1,"ABCDEF")
SetTextPosition(1,50,50)
Sync()
rem *** Calculate an angle step size for each
character ***
for c = 0 to 5
    rem *** select a random angle ***
    angle = Random(1,360)
    rem *** Calculate 40th of angle ***
    anglesteps#[c] = angle / 40.0
next c
do
    rem *** If pointer over text drop first letter
    ⚡remaining ***
    if GetTextHitTest(1,GetPointerX(),GetPointerY())
        rem *** FOR 40 times DO ***
        for c = 1 to 40
            for ch = 0 to 5
                rem *** Move the down 1% ***
                SetTextCharY(1,ch,c)
                rem *** Increase angle ***
                angle# = anglesteps#[ch]*c
                SetTextCharAngle(1,ch,angle#)
                Sync()
            next ch
        next c
    endif
loop
```

The problem with this example is that the falling process is rather slow.

We could improve the speed by giving a step size of 2 or 4 to the *for c = 1 to 40* loop.

Activity 14.11

Moving the pointer over the fallen characters has no effect, but moving it over the original position starts the whole process running once more with the characters falling from their original position.

Activity 14.12

Modified code for *ColourfulCharacters*:

```
rem *** Colour characters ***
rem *** Create and size text ***
CreateText(1,"oooooooooo")
SetTextSize(1,10)
do
    rem *** FOR each character DO ***
    for c = 0 to 9
        rem *** Change to random colour ***
        SetTextCharColor(1,c,Random(1,255),
            ⚡Random(1,255),Random(1,255),255)
        Sync()
        Sleep(200)
        rem *** Change back to white ***
        SetTextCharColor(1,c,255,255,255,255)
        Sync()
    next c
loop
```

```
next c
loop
```

Activity 14.13

When the characters are displayed initially they use the smooth but blurred style. Adding the new line to the code changes this to the sharper but blocky style.

Activity 14.14

No solution required.

15

User Input

In this Chapter:

- ☐ Virtual Buttons
- ☐ Keyboard Input
- ☐ Using Edit Boxes
- ☐ Virtual Joysticks
- ☐ Physical Joysticks
- ☐ Device-Specific Input
- ☐ Identifying a Device's Operating System

Virtual Buttons

Introduction

We have already made use of the AGK **virtual buttons** in several programs in earlier chapters. It is these buttons that appear when you called the *SetUpButtons()* function supplied in an earlier chapter. Each virtual button is assigned an image automatically (although that image can be replaced). A typical virtual button is shown in FIG-15.1.

FIG-15.1

A Typical Virtual Button



In this section we will examine the AGK statements used to create and manipulate those buttons.

Virtual Button Statements

AddVirtualButton()

To create a virtual button we need to use the `AddVirtualButton()` statement. This not only assigns an ID to the button, but also positions and sizes the button. The format of the statement is given in FIG-15.2.

FIG-15.2

AddVirtualButton()

`AddVirtualButton` (`id` , `x` , `y` , `fwidth`)

where:

- | | |
|---------------|---|
| id | is an integer value specifying the ID to be assigned to the virtual button. |
| x,y | are real values giving the coordinates at which the button is to be positioned. These coordinates represent the position of the centre of the button. |
| fwidth | is a real number giving the width of the button (percentage or virtual pixels). |

Activity 15.1

Start a new project named *UsingVirtualButtons* and write code to create a single button in the centre of the screen with a width setting of 10.

What happens when you click/press on the button? Save your project.

SetVirtualButtonText()

You can add text to the button using the `SetVirtualButtonText()` statement (see FIG-15.3).

FIG-15.3

SetVirtualButtonText()

`SetVirtualButtonText` (`id` , `string`)

where:

id is an integer value specifying the ID assigned to the button.

string is a string value giving the text to be placed within the button.

Activity 15.2

In *UsingVirtualButtons*, place the text *Yes* in the button.

Test and save your project.

The size of the text is fixed automatically, so if you try to assign a string with too many characters, it will overflow the edges of the button.

SetVirtualButtonColor()

FIG-15.4

You can change the colour of the button using the `SetVirtualButtonColor()` statement (see FIG-15.4).

SetVirtualButtonColor()

`SetVirtualButtonColor ((id , ired , igreen , iblue)`

where:

id is an integer value specifying the ID assigned to the button.

ired is an integer value giving the intensity of the red component of the colour (0: no red; 255: full red).

igreen is an integer value giving the intensity of the green component of the colour (0: no green; 255: full green).

iblue is an integer value giving the intensity of the blue component of the colour (0: no blue; 255: full blue).

Activity 15.3

In *UsingVirtualButtons*, set the colour of the button to yellow (R:255, G:255, B:0).

Test and save your project.

SetVirtualButtonAlpha()

A button's transparency level can be set using the `SetVirtualButtonAlpha()` statement (see FIG-15.5).

FIG-15.5

SetVirtualButtonAlpha()

`integer SetVirtualButtonAlpha ((id , itrans)`

where:

id is an integer value specifying the ID assigned to the button.

itrans is an integer value giving the transparency setting of the button (0 : invisible, 255: opaque).

Activity 15.4

In *UsingVirtualButtons*, make the button translucent with a setting of 126. Test and save your project.

SetVirtualButtonPosition()

A button can be repositioned at any time using the `SetVirtualButtonPosition()` statement (see FIG-15.6)

FIG-15.6

SetVirtualButtonPosition()

`SetVirtualButtonPosition ((id , x , y))`

where:

id is an integer value specifying the ID assigned to the button.

x,y are real values giving the coordinates of the centre of the button.

SetVirtualButtonSize()

A button can also be resized using the `SetVirtualButtonSize()` statement (see FIG-15.7)

FIG-15.7

SetVirtualButtonSize()

`SetVirtualButtonSize ((id , fsize))`

where:

id is an integer value specifying the ID assigned to the button.

fsize is a real value giving the new width of the button.

SetVirtualButtonVisible()

A button's visibility can be changed using the `SetVirtualButtonVisible()` statement (see FIG-15.8)

FIG-15.8

SetVirtualButtonVisible()

`SetVirtualButtonVisible ((id , iv))`

where:

id is an integer value specifying the ID assigned to the button.

iv is an integer value (0 or 1) specifying the visibility of the button (0: invisible, 1: visible).

SetVirtualButtonActive()

A button can be made inactive, and hence unresponsive to user presses, with the `SetVirtualButtonActive()` statement (see FIG-15.9).

FIG-15.9

SetVirtualButtonActive()

integer `SetVirtualButtonActive ((id , iactive))`

where:

id is an integer value specifying the ID assigned to the button.

inactive is an integer value (0 or 1) specifying the button's response (0: inactive, 1: active).

Activity 15.5

Modify *UsingVirtualButtons*, so that the button becomes inactive after 5 seconds.

Test and save your project.

SetVirtualButtonImageUp() and SetVirtualButtonImageDown()

We have seen how to set up a virtual button and add text to it, but a better option is to load your own images onto the button. Two images are used; one for the unpressed or “up” version of the button, the other for the pressed “down” version.

These images are loaded onto the button using the `SetVirtualButtonImageUp()` and `SetVirtualButtonImageDown()` statements (see FIG-15.10 and FIG-15.11).

FIG-15.10

SetVirtualButtonImageUp()

`SetVirtualButtonImageUp (id , imgId)`

where:

id is an integer value specifying the ID assigned to the button.

imgId is an integer value giving the ID of the image to be loaded. This image will be displayed when the button is unpressed.

FIG-15.11

SetVirtualButtonImageDown()

`SetVirtualButtonImageDown (id , imgId)`

where:

id is an integer value specifying the ID assigned to the button.

imgId is an integer value giving the ID of the image to be loaded. This image will be displayed when the button is pressed.

The program in FIG-15.12 displays a button showing the Japanese Hiragana symbol for the letter “a” using a slightly different image for the up and down positions.

FIG-15.12

Using Images on a Virtual Button

```
rem *** Using Virtual Buttons 2 ***

rem *** Load images used ***
LoadImage (1,"AUp.png",0)
LoadImage (2,"ADown.png",0)
rem *** Create button ***
AddVirtualButton (1,50,90,10)
rem *** Add images to button ***
SetVirtualButtonImageUp (1,1)
SetVirtualButtonImageDown (1,2)
do
  Sync ()
loop
```

Activity 15.6

Create a new project called *VB2*. Copy the files *AUp.png* and *ADown.png* from *AGKDownloads/Chapter15* then implement the code in FIG-15.12.

Test and save your project.

GetVirtualButtonPressed()

Although we have created a button, we also need to get the program to react to that button being pressed. We can check if a button is pressed (and then execute other code) using the `GetVirtualButtonPressed()` statement (see FIG-15.13).

FIG-15.13

`GetVirtualButtonPressed()` integer `GetVirtualButtonPressed` (`id`)

where:

id is an integer value specifying the ID assigned to the button.

This function returns 1 at the instant the virtual button is first pressed. At all other times zero is returned, irrespective of the button being up or down.

Activity 15.7

In your *VB2* project, get the program to add the letter *a* to a displayed string each time the virtual button is pressed. This requires the following additional lines of code:

```
rem *** Create text object ***
text$ = ""
CreateText(1,text$)
SetTextPosition(1,40,50)

rem *** IF key presses, add an "a" ***
if GetVirtualButtonPressed(1)=1
    text$=text$+"a"
    SetTextString(1,text$)
endif
```

Test and save your program.

GetVirtualButtonReleased()

You can also detect the moment a button is released using the `GetVirtualButtonReleased()` statement (see FIG-15.14).

FIG-15.14

`GetVirtualButtonReleased()` integer `GetVirtualButtonReleased` (`id`)

where:

id is an integer value specifying the ID assigned to the button.

The function returns 1 only at the instant the button is released, at all other times, zero is returned.

GetVirtualButtonState()

Since `GetVirtualButtonPressed()` and `GetVirtualButtonReleased()` only return 1 for a single instant, neither is useful for giving you details of the current state of a button. To discover if a button is currently being held down or is in the released state, the `GetVirtualButtonState()` command can be used (see FIG-15.15).

FIG-15.15

`GetVirtualButtonState()` integer `GetVirtualButtonState` ((`id`))

where:

id is an integer value specifying the ID assigned to the button.

The function returns 1 when the button is being held down and zero when the button is untouched.

GetVirtualButtonExists()

To check if a button of a given ID currently exists, the `GetVirtualButtonExists()` statement is used. This has the format shown in FIG-15.16.

FIG-15.16

`GetVirtualButtonExists()` integer `GetVirtualButtonExists` ((`id`))

where:

id is an integer value specifying the ID to be checked.

The function returns 1 if a virtual button of the specified ID exists, otherwise zero is returned.

DeleteVirtualButton()

When a button is no longer required, it can be deleted using the `DeleteVirtualButton()` statement (see FIG-15.17).

FIG-15.17

`DeleteVirtualButton()` `DeleteVirtualButton` ((`id`))

where:

id is an integer value specifying the ID of the button to be deleted.

Using Multiple Virtual Buttons

A maximum of 12 virtual buttons can exist at any one time in a program. When more than one button exists, we need to cycle through each button checking to see if it has been pressed. This requires code which implements the following logic:

```
FOR each button DO
  IF button just pressed THEN
    Execute code associated with that button
  ENDIF
ENDFOR
```

The program in FIG-15.18 is an extension of the code in project *VB2*. This time there are five Japanese character keys. These represent the vowels a, i, u, e, o in that order.

FIG-15.18

Using Images on Virtual Buttons

```

rem *** Using Virtual Buttons 3 ***

rem *** Global variables - image IDs ***
global AUp, ADown, IUp, IDown, UUp, Udown, EUp, EDown, OUp, ODown

rem *** Main logic ***
LoadImages()
SetUpButtons()
rem *** Create text object ***
text$ = ""
CreateText(1,"")
SetTextPosition(1,40,50)
do
    rem *** IF key pressed, add appropriate vowel ***
    for c = 1 to 5
        if GetVirtualButtonPressed(c)=1
            text$=text$+Mid("aiueo",c,1)
            SetTextString(1,text$)
        endif
    next c
    Sync()
loop

rem *** Functions ***
function LoadImages()
    AUp = LoadImage("AUp.png")
    ADown = LoadImage("ADown.png")
    IUp = LoadImage("IUp.png")
    IDown = LoadImage("IDown.png")
    UUp = LoadImage("UUp.png")
    UDown = LoadImage("UDown.png")
    EUp = LoadImage("EUp.png")
    EDown = LoadImage("EDown.png")
    OUp = LoadImage("OUp.png")
    ODown = LoadImage("ODown.png")
endfunction

function SetUpButtons()
    rem *** A ***
    AddVirtualButton(1,28,90,10)
    SetVirtualButtonImageUp(1,AUp)
    SetVirtualButtonImageDown(1,ADown)
    rem *** I ***
    AddVirtualButton(2,39,90,10)
    SetVirtualButtonImageUp(2,IUp)
    SetVirtualButtonImageDown(2,IDown)
    rem *** U ***
    AddVirtualButton(3,50,90,10)
    SetVirtualButtonImageUp(3,UUp)
    SetVirtualButtonImageDown(3,UDown)
    rem *** E ***
    AddVirtualButton(4,61,90,10)
    SetVirtualButtonImageUp(4,EUp)
    SetVirtualButtonImageDown(4,EDown)
    rem *** O ***
    AddVirtualButton(5,72,90,10)
    SetVirtualButtonImageUp(5,AUp)
    SetVirtualButtonImageDown(5,ADown)
endfunction

```

Activity 15.8

Start a new project called *HiraganaButtons*. Copy the appropriate image files from *AGKDownload/Chapter15* to the project's *media* folder then copy the code given in FIG-15.18 into *main.agc*.

Test the code to check that the buttons add the correct characters to the displayed string. Save your project.

Summary

- Use `AddVirtualButton()` to create up to 12 virtual buttons. The command also positions and sizes a button.
- Virtual buttons are automatically assigned “up” and “down” images which are displayed when the button is unpressed (up) or pressed (down).
- Use `SetVirtualButtonText()` to place text on a button.
- Use `SetVirtualButtonColor()` to colour the button image.
- Use `SetVirtualButtonAlpha()` to set the button's transparency.
- Use `SetVirtualButtonPosition()` to reposition a button.
- Use `SetVirtualButtonSize()` to resize a button.
- Use `SetVirtualButtonVisible()` to hide or display a button.
- Use `SetVirtualButtonActive()` to deactivate/activate a button.
- Use `SetVirtualButtonImageUp()` to replace the image used when a button is unpressed.
- Use `SetVirtualButtonImageDown()` to replace the image used when a button is pressed.
- Use `GetVirtualButtonPressed()` to detect the instant a button is pressed.
- Use `GetVirtualButtonReleased()` to detect the instant a button is released.
- Use `GetVirtualButtonState()` to determine if a button is currently pressed or unpressed.
- Use `GetVirtualButtonExists()` to check that a button of a specified ID currently exists.
- Use `DeleteVirtualButton()` to delete an existing button.

Keyboard Input

Introduction

Virtual buttons and sprites are fine for user input as long as that input is relatively simple, but when we need the user to enter alphanumeric values, then a full keyboard is required. This can be achieved by using a text-input resource. Adding a text-input resource automatically creates an on-screen edit box and, when your app is running on a tablet or smartphone, a virtual keyboard (see FIG-15.19).

FIG-15.19

A Typical Keyboard Input
Screen



The commands associated with a text-input resource are explained below.

Text-Input Statements

StartTextInput()

Many programming languages contain commands which will cause a program to halt while the user enters information at a keyboard. And while this is not a problem for traditional data entry situations, such a setup does not always suit a gaming environment where we may wish animations or other activities to continue in the background while data is keyed in.

To solve this problem, AGK splits keyboard entry into several stages. The first of these begins when the `StartTextInput()` statement is executed.

This statement creates a rectangular area on the screen for text entry. It also causes a virtual keyboard to appear when a hardware keyboard is not attached to the device. The statement has the format shown in FIG-15.20.

FIG-15.20

StartTextInput()

StartTextInput ()

The program in FIG-15.21 shows how minimum code is necessary to initiate keyboard input.

FIG-15.21

Using StartTextInput()

```
rem *** Keyboard input ***

StartTextInput ()
do
    Sync ()
loop
```

Activity 15.9

Start a new project called *KeyboardInput* and copy the code given above into *main.agc*.

Test and save the project.

GetTextInputCompleted()

The statement `GetTextInputCompleted()` will return 1 at the instant text input has been completed or cancelled (when the *Enter* or *Esc* key is pressed). At all other times the statement returns zero. The format for this command is given in FIG-15.22.

FIG-15.22

GetTextInputCompleted() integer `GetTextInputCompleted()` ()

Activity 15.10

Modify *KeyboardInput* so that the word “Completed” appears when the *Enter* key is pressed. (HINT: Make use of a text object to produce the word.)

Test and save the project.

GetTextInput()

When `GetTextInputCompleted()` returns a 1, you can then retrieve the string that was entered by the user using the `GetTextInput()` statement (see FIG-15.23).

FIG-15.23

GetTextInput() string `GetTextInput()` ()

The function returns the string entered.

Activity 15.11

Modify *KeyboardInput* so that in place of the word *Completed*, the program displays the text entered when input has been completed.

Test and save the project.

GetTextInputCancelled()

The user can decide to cancel text input by pressing the *Escape* key. If this happens, then the statement `GetTextInputCancelled()` will return 1. On the other hand, if the player enters data normally and finishes by pressing the *Enter* key, then `GetTextInputCancelled()` returns zero.

FIG-15.24

`GetTextInputCancelled()` has the format shown in FIG-15.24.

GetTextInputCancelled() integer `GetTextInputCancelled()` ()

This statement can only be called after `GetTextInputCompleted()` has returned 1.

Activity 15.12

Modify *KeyboardInput* so that, if keyboard entry is cancelled by the user, the message *User cancelled* is displayed.

Test and save your project.

StopTextInput()

Although the text input box and virtual keyboard will automatically be removed as soon as the user presses the *Enter* or *Escape* key, it is also possible to achieve the same effect with your code using the `StopTextInput()` statement.

An obvious reason to do this would be that the *Enter* key has not been pressed after a reasonable time period.

The `StopTextInput()` statement's format is shown in FIG-15.25.

FIG-15.25

StopTextInput()

StopTextInput ()

Activity 15.13

Modify *KeyboardInput* so that keyboard entry is cancelled if the *Enter* or *Escape* key is not pressed within 8 seconds.

Test and save your project.

GetTextInputState()

Whereas `GetTextInputCompleted()` will return 1 only at the instant the *Enter* or *Escape* key is first pressed, `GetTextInputState()` returns a value of zero during the whole text input stage and always returns 1 when no text input is being requested. Hence, this statement returns 1 if executed before `StartTextInput()` and also returns 1 at any time after the *Enter* (or *Escape*) key has been pressed.

FIG-15.26

GetTextInputState()

integer GetTextInputState ()

The program in FIG-15.27 highlights the use of the `GetTextInputState()` command by displaying a message to indicate the value being returned by the statement at various points in the program.

FIG-15.27

Using GetTextInputState()

```
rem *** Set up blank text ***
CreateText(1,"")
rem *** Save current time ***
time = GetSeconds()
rem *** No text entry at this point ***
if GetTextInputState() = 1
    SetTextString(1,"No text entry")
    Sync()
endif
```



FIG-15.27

(continued)

Using `GetTextInputState()`

```

rem *** Accept text after 6 seconds ***
repeat
    if GetSeconds() - time = 6
        StartTextInput()
    endif
    Sync()
until GetSeconds() - time = 7
rem *** Display text entry state message ***
do
    if GetTextInputState() = 0
        SetTextString(1,"Text being entered")
    else
        SetTextString(1,"No text being entered")
    endif
    Sync()
loop

```

Activity 15.14

Start a new project called *KeyboardState* and implement the code given in FIG-15.27.

Test and save your program.

SetTextInputMaxChars()

If you want to impose a limit on the number of characters that can be entered when using `StartTextInput()`, use the `SetTextInputMaxChars()` statement (see FIG-15.28).

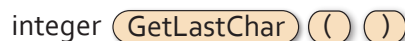
FIG-15.28`SetTextInputMaxChars()`


where

imax is an integer value giving the maximum characters allowed. Use zero if you want unlimited characters.

GetLastChar()


The ASCII code for the last character to be entered when using the `StartTextInput()` statement can be retrieved using `GetLastChar()` (see FIG-15.29).

FIG-15.29`GetLastChar()`


The function returns zero if no character has been entered.

SetCursorBlinkTime()

A final attribute you can control when using `StartTextInput()` is the cursor flash rate. This is done using `SetCursorBlinkTime()` (see FIG-15.30).

FIG-15.30`SetCursorBlinkTime()`


where

fsecs is a real number giving the interval between cursor blinks in fractions of a second.

The default blink time is about 0.5 seconds.

Summary

- For a full range of character input we can use a text-input resource.
- Use `StartTextInput()` to activate a text-input resource.
- Once started, text-input automatically creates a text input box and, where no physical keyboard is connected, a virtual, on-screen keyboard.
- Use `GetTextInputCompleted()` to determine if the user has pressed the *Enter* or *Escape* key.
- Use `GetTextInputCancelled()` to determine if the user has aborted the input operation by pressing the *Escape* key.
- When text input has been completed, use `GetTextInput()` to determine the string entered by the user.
- Use `StopTextInput()` to terminate text input via your program code.
- Use `GetTextInputState()` to determine if the program is currently accepting text input from the user.
- Use `SetTextInputMaxChars()` to specify a maximum number of characters to be allowed within the edit box.
- Use `GetLastChar()` to find the ASCII code for the last character entered at the keyboard.
- Use `SetCursorBlinkTime()` To set the flash rate for the cursor within the edit box.

Edit Box Statements

Introduction

The text input commands are fine if all we want is to get something simple like a player's name or a password, but it isn't really suitable if we want to gather more complex information.

For complete control over the number and position of data entry edit boxes, AGK has a set of edit box commands.

A typical screen layout produced using these commands is shown in FIG-15.31.

FIG-15.31

A Simple Edit Box Layout



Edit Box Statements

CreateEditBox()

A new edit box can be created using the `CreateEditBox()` statement (see FIG-15.32).

FIG-15.32

`CreateEditBox()`

Format 1

`CreateEditBox (id)`

Format 2

`integer CreateEditBox ()`

where

id is an integer value specifying the ID to be assigned to the edit box. No two edit boxes may be assigned the same ID value.

Format 1 allows you to specify the ID to be assigned; format 2 returns the ID automatically assigned by AGK.

SetEditBoxSize()

FIG-15.33

SetEditBoxSize()

To size an edit box, use `SetEditBoxSize()` (see FIG-15.33).

`SetEditBoxSize (id , fwidth , fheight)`

where

id is an integer value giving the ID previously assigned to the edit box.

fwidth is a real value giving the width of the edit box (percentage or virtual coordinates as appropriate).

fheight is a real value giving the height of the edit box.

The dimensions given are for the inside of the edit box. The border takes up additional space.

GetEditBoxWidth() and GetEditBoxHeight()

FIG-15.34

GetEditBoxWidth()

GetEditBoxHeight()

The current dimensions of an edit box can be found using the `GetEditBoxWidth()` and `GetEditBoxHeight()` statements (see FIG-15.34).

`float GetEditBoxWidth (id)`

`float GetEditBoxHeight (id)`

where

id is an integer value giving the ID of the edit box whose dimensions are to be found.

The value returned by each function will be a percentage of the screen size or in virtual pixels depending on the system your program has been set up for.

SetEditBoxPosition()

FIG-15.35

SetEditBoxPosition()

To position an edit box once it has been created, use `SetEditBoxPosition()` (see FIG-15.35).

`SetEditBoxPosition (id , x , y)`

where

id is an integer value giving the ID previously assigned to the edit box.

x,y

are real values giving the coordinates at which the top-left corner of the edit box is to be positioned.

GetEditBoxX() and GetEditBoxY()

The coordinates of the edit box's top-left corner can be found using `GetEditBoxX()` and `GetEditBoxY()` (see FIG-15.36).

FIG-15.36

`GetEditBoxX()`

`GetEditBoxY()`

float `GetEditBoxX (id)`

float `GetEditBoxY (id)`

where

id

is an integer value giving the ID of the edit box whose position is to be found.

SetEditBoxMaxChars()

The maximum number of characters that can be entered within an edit box is set using `SetEditBoxMaxChars()` (see FIG-15.37).

FIG-15.37

`SetEditBoxMaxChars()`

`SetEditBoxMaxChars (id , imax)`

where

id

is an integer value giving the ID previously assigned to the edit box.

imax

is an integer value giving the maximum characters allowed within the edit box.

GetEditBoxText()

FIG-15.38

`GetEditBoxText()`

To access the text entered in an edit box, use `GetEditBoxText()` (see FIG-15.38).

string `GetEditBoxText (id)`

where

id

is an integer value giving the ID previously assigned to the edit box.

The function returns a string containing a copy of the text within the specified edit box.

SetEditBoxFocus()

An edit box is said to have **focus** when it has been selected and you see the cursor within the the box awaiting user input. You can give an edit box focus simply by clicking within the box or you can assign focus using the `SetEditBoxFocus()` statement (see FIG-15.39).

FIG-15.39

`SetEditBoxFocus()`

`SetEditBoxFocus (id , ifocus)`

where

- id** is an integer value giving the ID previously assigned to the edit box.
- ifocus** is an integer value (0 or 1) which either assigns focus to the edit box (1) or removes focus (0).

An edit box can lose focus in several ways: the user presses the **Enter** key to complete text entry within the box, the user clicks on another edit box, or a program command assigns focus to a different edit box.

GetEditBoxHasFocus()

To check if a specific edit box currently has focus, use `GetEditBoxHasFocus()` (see FIG-15.40).

FIG-15.40

GetEditBoxHasFocus()

integer `GetEditBoxHasFocus` ((id))

where

- id** is an integer value giving the ID previously assigned to the edit box.

The function returns 1 if the specified edit box has focus, otherwise zero is returned.

GetCurrentEditBox()

To discover the ID of the edit box currently with focus, use `GetCurrentEditBox()` (see FIG-15.41).

FIG-15.41

GetCurrentEditBox()

integer `GetCurrentEditBox` ()

The function returns the ID of the edit box which currently has focus. If no edit box has focus, zero is returned.

GetEditBoxChanged()

You can check if an edit box has just lost focus using `GetEditBoxChanged()` (see FIG-15.42). The function returns 1 at the moment the edit box loses focus.

FIG-15.42

GetEditBoxChanged()

integer `GetEditBoxChanged` ((id))

where

- id** is an integer value giving the ID of the edit box to be checked.

If we assume that when an edit box loses focus its content will have been changed, then this function is useful for initiating any process you wish to carry out because of these changes.

The program in FIG-15.43 demonstrates the use of most of the edit box statements already described by creating a single edit box, setting the maximum characters allowed to 15, assigning it focus, and when focus is lost, displaying the contents of the edit box.

FIG-15.43

Using Edit Boxes

```

rem *** Using Edit boxes ***

rem *** Create the edit box ***
CreateEditBox(1)
SetEditBoxSize(1,70,5)
SetEditBoxPosition(1,10,20)
rem *** Set maximum characters to 15 ***
SetEditBoxMaxChars(1,15)
rem *** Assign focus to the edit box ***
SetEditBoxFocus(1,1)
do
    rem *** If the box loses focus, display contents ***
    if GetEditBoxChanged(1) = 1
        Print("Text entered was : "+GetEditBoxText(1))
    endif
    Sync()
loop

```

Activity 15.15

Start a new project called *EditBox01* and implement the code given in FIG-15.43.

Test your program. Check what happens when you try to enter more than 15 characters.

Save your project.

SetEditBoxTextSize()

The text within a text box is automatically set to be two units less than the height of the box and there will be times when this is too large. You can change the height of the text using the `SetEditBoxTextSize()` statement (see FIG-15.44).

FIG-15.44

SetEditBoxTextSize()

`SetEditBoxTextSize` (`id` , `fsz`)

where

id is an integer value giving the ID previously assigned to the edit box.

fsz is a real value giving the height of the text (percentage or virtual pixels, as appropriate).

Activity 15.16

Modify *EditBox01* so that the text size used within the edit box is set to 3.5.

Modify the code again so that there is no limit on the number of characters that can be entered in the edit box.

What happens when the edit box has been filled?

Save your project.

SetEditTextColor()

The colour of the text appearing within an edit box can also be specified using the `SetEditTextColor()` statement (see FIG-15.45).

FIG-15.45

SetEditTextColor()

`SetEditTextColor ((id , ired , igreen , iblue)`

where

id	is an integer value giving the ID previously assigned to the edit box.
ired	is an integer (0 to 255) value giving the intensity of the red component of the text's colour (0: no red, 255: full red).
igreen	is an integer (0 to 255) value giving the intensity of the green component of the text's colour (0: no green, 255: full green).
iblue	is an integer (0 to 255) value giving the intensity of the blue component of the text's colour (0: no blue, 255: full blue).

SetEditBoxFontImage()

To change the font used for the text that appears within the edit box, use `SetEditBoxFontImage()` (see FIG-15.46).

FIG-15.46

SetEditBoxFontImage()

`SetEditBoxFontImage ((id , imgId)`

where

Details on how to create a monospaced font image was described in Chapter 14. Details of how to create a proportional font image are covered in Chapter 16.

id	is an integer value giving the ID previously assigned to the edit box.
imgId	is an integer value giving the ID of the previously loaded image containing the required font.

SetEditBoxMultiLine()

An edit box can be modified to accept multiple lines of text - useful if you want the user to enter details such as their postal address or a lengthy comment.

To allow an edit box to accept multiple lines, use the `SetEditBoxMultiLine()` statement (see FIG-15.47).

FIG-15.47

SetEditBoxMultiLine()

`SetEditBoxMultiLine ((id , iop)`

where

id	is an integer value giving the ID previously assigned to the edit box.
iop	is an integer value (0 or 1) which specifies if multi-line entry is allowed (1) or disabled (0).

A multi-line edit box will move to a new line when the current line is full or the **Enter**

key is pressed. Because it accepts the *Enter* key as part of the input, the edit box will only lose focus when the user selects an area outside that edit box.

SetEditBoxMaxLines()

Where multi-line entry is allowed, the maximum number of lines the user can enter can be specified using `SetEditBoxMaxLines()` (see FIG-15.48).

FIG-15.48

SetEditBoxMaxLines()

`SetEditBoxMaxLines (id , imax)`

where

id is an integer value giving the ID previously assigned to the edit box.

imax is an integer value giving the maximum number of lines allowed. A value of zero gives unlimited lines.

When using multiple lines, the height of the box and the text within it should be adjusted accordingly, otherwise the lines of text will scroll within the box.

Activity 15.17

Modify *EditBox01* so that a second edit box is added. This box should be sized as 70x20, positioned at (10,30), and be enabled for a maximum of 7 lines. Change the text sizes in each edit box to 2.5.

When you press *Enter* to end text entry in the first edit box, focus does not shift automatically to the second box. Modify your code so that it does and remove the code to display the contents of the first box.

Test and save your project.

GetEditBoxLines()

To discover the number of lines of text the user has entered in an edit box, use `GetEditBoxLines()` (see FIG-15.49).

FIG-15.49

GetEditBoxLines()

integer `GetEditBoxLines (id)`

where

id is an integer value giving the ID of the edit box whose lines setting is to be found.

An edit box in which no text has been entered will return 0.

SetEditBoxText()

Any realistic form has to have some sort of indication of what the user is expected to enter in each box. We could achieve this by adding text objects above or to the side of each edit box. But a quicker way is to include such details within the edit box as its initial text contents. This can be done using the `SetEditBoxText()` statement (see FIG-15.50).

FIG-15.50

SetEditText()


 The diagram shows the function signature `SetEditText` followed by its parameters in parentheses: `(id , svalue)`. The `id` and `svalue` identifiers are highlighted with green boxes.

where

id is an integer value giving the ID previously assigned to the edit box.

svalue is a string giving the initial text to appear within the edit box.

When an edit box gains focus, any text set up by the `SetEditText()` statement will remain and new text is added to the end of that existing text. To get rid of the text just as the user begins to type, we need execute another `SetEditText()` statement setting the contents of the box to an empty string.

The program in FIG-15.51 is an extension to the *EditBox01* adding initial statements to each edit box and clearing that text as soon as the box gains focus.

FIG-15.51Using Edit Box
Descriptors

```

rem *** Using Edit boxes ***

rem *** Initial text in edit boxes ***
dim initialtext[2] as string =["", "Name", "Postal Address"]

rem *** Create the name edit box ***
CreateEditBox(1)
SetEditBoxSize(1,70,5)
SetEditBoxPosition(1,10,20)
rem *** Set text size ***
SetEditBoxTextSize(1,2.5)

rem *** Set initial text ***
SetEditText(1,initialtext[1])

rem *** Create address edit box ***
CreateEditBox(2)
SetEditBoxSize(2,70,20)
SetEditBoxPosition(2,10,30)
rem *** Max 7 lines ***
SetEditBoxMultiLine(2,1)
SetEditBoxMaxLines(2,7)
rem *** Set text size ***
SetEditBoxTextSize(2,2.5)

rem *** initial text ***
SetEditText(2,initialtext[2])

do
  rem *** Get ID of current edit box ***
  id = GetCurrentEditBox()
  rem *** If an edit box is in focus ***
  if id <> 0
    rem *** If it contains its initial text ***
    if GetEditBoxText(id) = initialtext[id]
      rem *** Remove it ***
      SetEditBoxText(id,"")
    endif
  endif
  Sync()
loop

```

Activity 15.18

Modify *EditBox01* to match the code in FIG-15.51.

Test the program, checking that the descriptive text is removed when an edit box gains focus.

Modify the code so that, if an edit box loses focus when it contains no user-entered text, the original description is reinstated.

Test and save your project.

Various aspects of an edit box, such as the border, background and cursor can be modified using various commands.

FIG-15.52

SetEditBoxBackgroundColor()

SetEditBoxBackground
↳ Color()

To change the background colour within an edit box, use the `SetEditBoxBackgroundColor()` statement (see FIG-15.52).

```
SetEditBoxBackgroundColor ( ( id , ired , igreen , iblue , itrans )
```

where

id	is an integer value giving the ID previously assigned to the edit box.
ired	is an integer (0 to 255) value giving the intensity of the red component of the edit box's background colour (0: no red, 255: full red).
igreen	is an integer (0 to 255) value giving the intensity of the green component of the edit box's background colour (0: no green, 255: full green).
iblue	is an integer (0 to 255) value giving the intensity of the blue component of the edit box's background colour (0: no blue, 255: full blue).
itrans	is an integer value specifying the opacity of the background (0: invisible, 255: opaque).

SetEditBoxBackgroundImage()

FIG-15.53

If you want to have an image as the background within an edit box, you can use `SetEditBoxBackgroundImage()` (see FIG-15.53).

SetEditBoxBackground
↳ Image()

```
SetEditBoxBackgroundImage ( ( id , imgId )
```

where

id	is an integer value giving the ID previously assigned to the edit box.
-----------	--

imgId is an integer value giving the ID of the previously loaded image to be used as the edit box background.

To remove an existing image background, call the `SetEditBoxBackgroundImage()` statement with the *imgId* parameter set to zero.

Background colour and image can be combined, if you wish.

SetEditBoxBorderColor()

FIG-15.54

The edit box's border can also have its colour modified using the `SetEditBoxBorderColor()` statement (see FIG-15.54).

`SetEditBoxBorderColor ((id , ired , igreen , iblue , itrans)`

where

id is an integer value giving the ID previously assigned to the edit box.

ired is an integer (0 to 255) value giving the intensity of the red component of the edit box's border colour (0: no red, 255: full red).

igreen is an integer (0 to 255) value giving the intensity of the green component of the edit box's border colour (0: no green, 255: full green).

iblue is an integer (0 to 255) value giving the intensity of the blue component of the edit box's border colour (0: no blue, 255: full blue).

itrans is an integer value specifying the opacity of the border (0: invisible, 255: opaque).

SetEditBoxBorderImage()

FIG-15.55

A second option for modifying the edit box's border is to use an image. This requires the statement `SetEditBoxBorderImage()` (see FIG-15.55).

`SetEditBoxBorderImage ((id , imgId)`

where

id is an integer value giving the ID previously assigned to the edit box.

imgId is an integer value giving the ID of the previously loaded image to be used as the edit box border.

The border colour and image statements can be used in combination. To remove an existing border image, call `SetEditBoxBorderImage()` with the *imgId* parameter set to zero.

SetEditBoxBorderSize()

To change the width of the edit box border, use `SetEditBoxBorderSize()` (see FIG-15.56).

FIG-15.56

SetEditBoxBorderSize()

`SetEditBoxBorderSize ((id , fsize)`

where

id is an integer value giving the ID previously assigned to the edit box.

fsize is a real value giving the width of the border (percentage or virtual).

SetEditBoxCursorWidth()

The width of the cursor can be set using `SetEditBoxCursorWidth()` (see FIG-15.57).

FIG-15.57

SetEditBoxCursorWidth()

`SetEditBoxCursorWidth ((id , fwidth)`

where

id is an integer value giving the ID previously assigned to the edit box.

fwidth is a real value giving the width of the border (percentage or virtual). The default value is 1.5.

SetEditBoxCursorColor()

To change the colour of the edit box's cursor use `SetEditBoxCursorColor()` (see FIG-15.58).

FIG-15.58

SetEditBoxCursorColor()

`SetEditBoxCursorColor ((id , ired , igreen , iblue)`

where

id is an integer value giving the ID previously assigned to the edit box.

ired is an integer (0 to 255) value giving the intensity of the red component of the cursor's colour (0: no red, 255: full red).

igreen is an integer (0 to 255) value giving the intensity of the green component of the cursor's colour (0: no green, 255: full green).

iblue is an integer (0 to 255) value giving the intensity of the blue component of the cursor's colour (0: no blue, 255: full blue).

SetEditBoxCursorBlinkTime()

The final attribute of the cursor that can be modified is its blink time. You can do this using `SetEditBoxCursorBlinkTime()` (see FIG-15.59).

FIG-15.59

SetEditBoxCursorBlink
Time()

SetEditBoxCursorBlinkTime ((id , fsecs))

where

- id** is an integer value giving the ID previously assigned to the edit box.
- fsecs** is a real value giving the cursor's blink time in fractions of a second. The default time is about 0.5 seconds.

SetEditBoxScissor()

We can crop the part of an edit box which is actually visible on the screen using `SetEditBoxScissor()` (see FIG-15.60).

FIG-15.60

SetEditBoxScissor()

SetEditBoxScissor ((id , x1 , y1 , x2 , y2))

where

- id** is an integer value giving the ID previously assigned to the edit box.
- x1,y1** are real values giving the coordinates of the top-left corner of the visible area of the edit box.
- x2,y2** are real values giving the coordinates of the bottom-right corner of the visible area of the edit box.

In FIG-15.61 we see an edit box originally created using the code

```
CreateEditBox(2)
SetEditBoxSize(2,70,20)
SetEditBoxPosition(2,10,30)
```

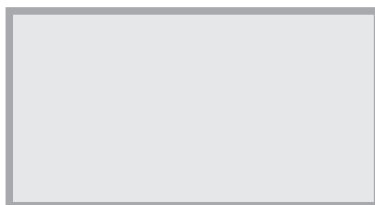
and then scissored using the line

```
SetEditBoxScissor(2,8,40,45,52)
```

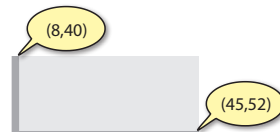
FIG-15.61

The Effect of Using
SetEditBoxScissor()

Original Edit Box



Scissored Edit Box



SetEditBoxActive()

There are occasions when we may want to stop the user entering data in a box. For example, if a form had a box labelled *Maiden Name*, we wouldn't want males or unmarried females to enter data there.

To make a visible box unable to gain focus, we can use the `SetEditBoxActive()` statement (see FIG-15.62).

FIG-15.62

SetEditBoxActive()

`SetEditBoxActive (id , iop)`

where

id is an integer value giving the ID previously assigned to the edit box.

iop is an integer (0 or 1) specifying whether the box is to be inactive (0) or active (1).

An inactive box cannot gain focus and hence the user cannot enter new data into that edit box; an active box can gain focus.

GetEditBoxActive()

The active state of an edit box can be determined using `GetEditBoxActive()` (see FIG-15.63).

FIG-15.63

GetEditBoxActive()

integer `GetEditBoxActive (id)`

where

id is an integer value giving the ID of the edit box to be checked.

The function returns 1 if the specified edit box is active, otherwise zero is returned.

SetEditBoxVisible()

An edit box can be made invisible (or made to reappear) using `SetEditBoxVisible()` (see FIG-15.64).

FIG-15.64

SetEditBoxVisible()

`SetEditBoxVisible (id , iv)`

where

id is an integer value giving the ID previously assigned to the edit box.

iv is an integer (0 or 1) specifying whether the box is to be visible (1) or invisible (0).

An invisible edit box cannot gain focus.

GetEditBoxVisible()

To check the visibility status of an edit box use `GetEditBoxVisible()` (see FIG-15.65).

FIG-15.65

GetEditBoxVisible()

integer `GetEditBoxVisible (id)`

where

id is an integer value giving the ID of the edit box to be checked.

The function returns 1 if the specified edit box is visible, otherwise zero is returned.

SetEditBoxDepth()

You may want to adjust the layer on which your edit box appears. By default, edit boxes, like sprites, are placed on layer 10. To modify the edit box's layer, use `SetEditBoxDepth()` (see FIG-15.66).

FIG-15.66

SetEditBoxDepth()

`SetEditBoxDepth ((id , idepth)`

where

id is an integer value giving the ID previously assigned to the edit box.

idepth is an integer value giving the layer on which the edit box is to be placed.

DeleteEditBox()

If an edit box is no longer required, it can be deleted using `DeleteEditBox()` (see FIG-15.67).

FIG-15.67

DeleteEditBox()

`DeleteEditBox ((id)`

where

id is an integer value giving the ID of the edit box to be deleted.

GetEditBoxExists()

To check that an edit box of a specified ID exists, use `GetEditBoxExists()` (see FIG-15.68).

FIG-15.68

GetEditBoxExists()

integer `GetEditBoxExists ((id)`

where

id is an integer value giving the ID of the edit box to be checked.

The function returns 1 if an edit box of the specified ID exists, otherwise zero is returned.

Summary

- Use `CreateEditBox()` to create an edit box.
- Use `SetEditBoxSize()` to set the dimensions of an edit box.
- Use `GetEditBoxWidth()` and `GetEditBoxHeight()` to discover the current dimensions of an edit box.
- Use `SetEditBoxPosition()` to position an edit box.

- Use `GetEditBoxX()` and `GetEditBoxY()` to find the current position of an edit box.
- Use `SetEditBoxMaxChars()` to set a maximum number of characters allowed within an edit box.
- Use `GetEditBoxText()` to retrieve the contents of an edit box.
- Use `SetEditBoxFocus()` to set/remove edit box focus.
- Use `GetEditBoxHasFocus()` to check if an edit box currently has focus.
- Use `GetCurrentEditBox()` to discover the ID of any edit box which has focus.
- Use `GetEditBoxChanged()` to detect if an edit box has just lost focus.
- Use `SetEditBoxTextSize()` to set the size of the text used within an edit box.
- Use `SetEditBoxTextColor()` to set the colour of the text used within an edit box.
- Use `SetEditBoxFontImage()` to change the text font used within an edit box.
- Use `SetEditBoxMultiLine()` to allow multiple lines of text to be entered within an edit box.
- Use `SetEditBoxMaxLines()` to set the maximum number of lines allowed within an edit box.
- Use `GetEditBoxLines()` to discover how many lines of text the user has entered in a multi-line edit box.
- Use `SetEditBoxText()` to set the text displayed within an edit box.
- Use `SetEditBoxBackgroundColor()` to set the colour used as background within an edit box.
- Use `SetEditBoxBackgroundImage()` to display an image as an edit box's background.
- Background colour and image can be used in combination.
- Use `SetEditBoxBorderColor()` to set the colour used in the border of an edit box.
- Use `SetEditBoxBorderImage()` to display an image in an edit box's border.
- Use `SetEditBoxBorderSize()` to set the width of an edit box's border.
- Use `SetEditBoxCursorWidth()` to set the width of the cursor which appears within an edit box.
- Use `SetEditBoxCursorColor()` to set the colour of the cursor within an edit box.
- Use `SetEditBoxCursorBlinkTime()` to set the flashing rate for an edit box's cursor.
- Use `SetEditBoxScissor()` to make only part of an edit box visible.
- Use `SetEditBoxActive()` to activate/deactivate an edit box.
- Use `GetEditBoxActive()` to discover if a specified edit box is currently active.
- Use `SetEditBoxVisible()` to make an edit box visible/invisible.

- Use `GetEditBoxVisible()` to discover if a specified edit box is currently visible.
- Use `SetEditBoxDepth()` to specify which layer an edit box is to be placed on.
- Use `DeleteEditBox()` to remove an edit box object from a program.
- Use `GetEditBoxExists()` to check if an edit box with a given ID currently exists.

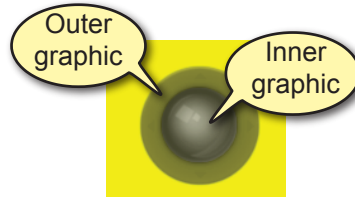
Joystick Input

Introduction

AGK can handle a real joystick or create a virtual one. Since most of you will be writing apps for portable devices, we'll start with the virtual joystick commands. These create a joystick-type interface on your screen which consists of two graphical components: a static outer graphic and a moveable inner graphic (see FIG-15.69).

FIG-15.69

The Default Virtual Joystick



Virtual Joystick Statements

AddVirtualJoystick()

To create a virtual joystick, we need to execute the `AddVirtualJoystick()` command, which also determines the position and size of the joystick. This statement has the format shown in FIG-15.70.

FIG-15.70

AddVirtualJoystick()

`AddVirtualJoystick` (`id` , `x` , `y` , `fsize`)

where:

- id** is an integer value giving the ID to be assigned to the joystick.
- x,y** are real numbers giving the coordinates at which the joystick is to be placed.
- fsize** is a real number giving the diameter of the joystick.

Activity 15.19

Start a new project called *VirtualJoystick* and code *main.agc* as:

```
rem *** Using a virtual joystick ***
rem *** Add joystick ***
SetClearColor(255,255,0)
AddVirtualJoystick(1,50,50,30)
do
    Sync()
loop
```

Test your program by dragging the centre of the joystick to see how it moves. Save your project.

SetVirtualJoystickPosition()

Your virtual joystick can be repositioned on the screen using the `SetVirtualJoystickPosition()` statement (see FIG-15.71).

FIG-15.71

SetVirtualJoystickPosition()


 A syntax diagram for the SetVirtualJoystickPosition function. It shows the function name in a rounded rectangle, followed by an opening parenthesis, the parameter 'id' in a green box, a comma, the parameter 'x' in a green box, a comma, the parameter 'y' in a green box, and a closing parenthesis.

where:

id is an integer value giving the ID assigned to the joystick.

x,y are real numbers giving the coordinates to which the joystick is to be moved.

Activity 15.20

Modify *VirtualJoystick* moving your joystick to the bottom right of the screen.

Test and save project.

SetVirtualJoystickSize()

Another change that can be made after the joystick has been created is to resize it using the `SetVirtualJoystickSize()` statement (see FIG-15.72).

FIG-15.72

SetVirtualJoystickSize()


 A syntax diagram for the SetVirtualJoystickSize function. It shows the function name in a rounded rectangle, followed by an opening parenthesis, the parameter 'id' in a green box, a comma, the parameter 'fsize' in a green box, and a closing parenthesis.

where:

id is an integer value giving the ID assigned to the joystick.

fsize is a real number giving the width of the joystick.

Activity 15.21

Modify *VirtualJoystick* changing the width of the joystick to 20.

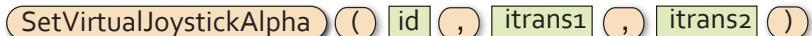
Test and save project.

SetVirtualJoystickAlpha()

You may have noticed that the outer ring of the joystick is translucent. You can modify the transparency of both the outer and inner parts of the joystick using the `SetVirtualJoystickAlpha()` statement (see FIG-15.73).

FIG-15.73

SetVirtualJoystickAlpha()


 A syntax diagram for the SetVirtualJoystickAlpha function. It shows the function name in a rounded rectangle, followed by an opening parenthesis, the parameter 'id' in a green box, a comma, the parameter 'itrans1' in a green box, a comma, the parameter 'itrans2' in a green box, and a closing parenthesis.

where:

id is an integer value giving the ID assigned to the joystick.

itrans1 is an integer value giving the transparency setting for the outer graphic. This value should be in the range 0 (invisible) to 255 (opaque).

itrans2 is an integer value giving the transparency setting for the inner graphic. This value should be in the range 0: invisible to 255: opaque.

Activity 15.22

Modify *VirtualJoystick* so that the inner and outer graphics of the joystick are opaque.

Test and save project.

SetVirtualJoystickImageInner() and SetVirtualJoystickImageOuter()

You can create your own images for the joystick's inner and outer parts. These must then be loaded using the standard `LoadImage()` statement before being assigned to the joystick using the `SetVirtualJoystickImageInner()` and `SetVirtualJoystickImageOuter()` statements (see FIG-15.74 and FIG-15.75).

FIG-15.74

SetVirtualJoystick-
ImageInner()

```
SetVirtualJoystickImageInner ( ( id , imgId ) )
```

where:

id is an integer value giving the ID assigned to the joystick.

imgId is an integer giving the ID of the image to be used for the inner part of the joystick.

FIG-15.75

SetVirtualJoystick-
ImageOuter()

```
SetVirtualJoystickImageOuter ( ( id , imgId ) )
```

where:

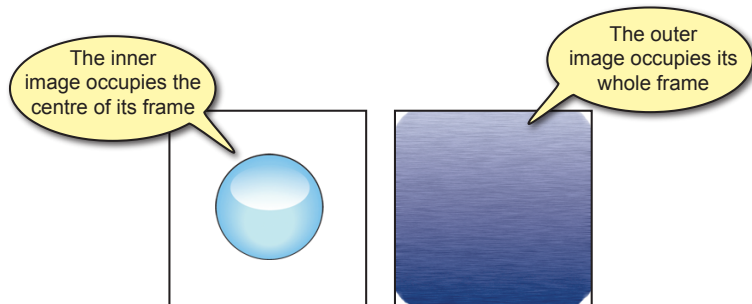
id is an integer value giving the ID assigned to the joystick.

imgId is an integer giving the ID of the image to be used for the outer part of the joystick.

When creating these two images, bear in mind that the inner image has to take up a smaller area within its own frame as shown in FIG-15.76.

FIG-15.76

Joystick's Inner and Outer
Images



Activity 15.23

Copy the files *JoystickOuter.png* and *JoystickInner.png* from *AGKDownloads/Chapter15* into *VirtualJoystick's media* folder.

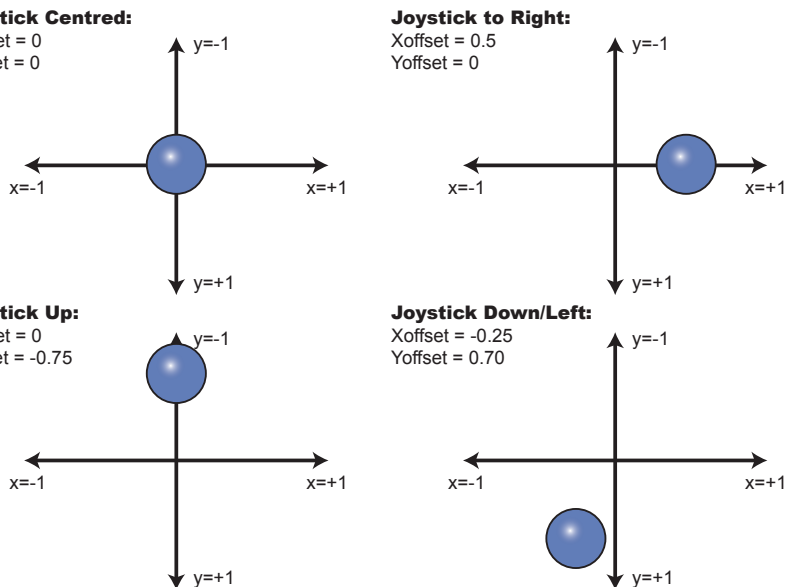
Use these two images to replace the default ones. Test and save your project.

GetVirtualJoystickX() and GetVirtualJoystickY()

When you move the joystick arm, you affect its offset in the x and y directions from its central position (see FIG-15.77).

FIG-15.77

Joystick Movement



To find the amount of offset applied in the x direction we can use the `GetVirtualJoystickX()` statement (see FIG-15.78). The offset in the y direction is given by `GetVirtualJoystickY()` (see FIG-15.79).

FIG-15.78

`GetVirtualJoystickX()`

float `GetVirtualJoystickX` (`id`)

FIG-15.79

`GetVirtualJoystickY()`

float `GetVirtualJoystickY` (`id`)

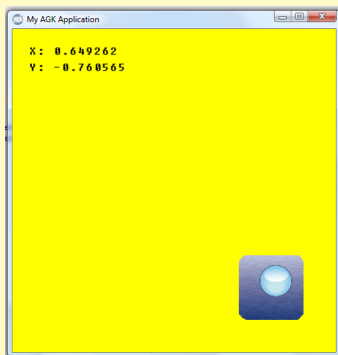
where:

id is an integer value giving the ID assigned to the joystick.

The values returned by these functions will lie in the range -1 to 1.

Activity 15.24

Modify *VirtualJoystick* so that it displays the x and y readings from the joystick. The display should look like that given below:



SetVirtualJoystickDeadZone()

When the joystick arm gets close to the centre position (it doesn't have to be exactly at the centre), `GetVirtualJoystickX()` and `GetVirtualJoystickY()` both return a reading of zero. This area close to the centre is known as the **dead zone**.

Activity 15.25

Rerun *VirtualJoystick* and find out what lowest absolute value can be achieved in both the *x* and *y* directions before the value zero is displayed.

In theory, at least, you should be able to get a value of around 0.15 when moving the joystick before there is a sudden jump to zero. 0.15 is the default value used.

Using the `SetVirtualJoystickDeadZone()` statement, we can adjust the area of the dead zone for all virtual joysticks by setting the smallest offset possible before zero is returned. This statement has the format shown in FIG-15.80.

FIG-15.80

SetVirtualJoystickDeadZone(`SetVirtualJoystickDeadZone` (`fzone`)

where:

fzone is a real number giving the smallest absolute value detectable using the joystick. This should be in the range 0 to 1.

Activity 15.26

Modify *VirtualJoystick* so that a value as low as 0.08 can be detected.

Test and save your project.

The `SetVirtualJoystickDeadZone()` statement sets the dead zone for all joysticks being used.

SetVirtualJoystickActive()

You can make a joystick unresponsive to the user (or reactivate it again) using the `SetVirtualJoystickActive()` statement (see FIG-15.81).

FIG-15.81

SetVirtualJoystickActive(`SetVirtualJoystickActive` (`id` , `iactive`)

where:

id is an integer value giving the ID assigned to the joystick.

iactive is an integer value (0 or 1). A value of zero makes the specified joystick inactive; 1 reactivates it.

Activity 15.27

Modify *VirtualJoystick* so that the joystick becomes inactive after 5 seconds and reactivates after 8.

Test and save your project.

In a game environment, you'll probably want to use a joystick to move an object on the screen. The program in FIG-15.82 uses the joystick to move the crosshairs of a weapon sight.

FIG-15.82

Using a Virtual Joystick

```
rem *** Using a virtual joystick ***
rem *** Load images required ***
LoadImage(1,"JoystickOuter.png",0)
LoadImage(2,"JoystickInner.png",0)
LoadImage(3,"Crosshairs.png",0)
rem *** Create and position crosshairs sprite ***
CreateSprite(1,3)
SetSpriteSize(1,10,-1)
SetSpritePosition(1,45,45)
rem *** Add joystick ***
SetClearColor(255,255,0)
AddVirtualJoystick(1,50,50,30)
rem *** Change joystick images ***
SetVirtualJoystickImageOuter(1,1)
SetVirtualJoystickImageInner(1,2)
rem *** Reposition joystick ***
SetVirtualJoystickPosition(1,85,85)
rem *** Resize joystick ***
SetVirtualJoystickSize(1,20)
rem *** Make joystick opaque ***
SetVirtualJoystickAlpha(1,255,255)
do
    rem *** Update crosshairs' position ***
    SetSpritePosition(1, GetSpriteX(1)+ GetVirtualJoystickX(1) ,
        ↵GetSpriteY(1)+GetVirtualJoystickY(1) )
    Sync()
loop
```

The most important line in the program above is

```
SetSpritePosition(1,GetSpriteX(1)+GetVirtualJoystickX(1) ,
    ↵GetSpriteY(1)+GetVirtualJoystickY(1) )
```

which repositions the crosshairs using the values returned by the joystick.

Activity 15.28

Create a new project, *Crosshairs*, which implements the code given in FIG-15.82. Before testing the program, copy the file *AGKDownloads/Chapter15/Crosshairs.png*, *JoystickOuter.png* and *JoystickInner.png* into the project's *media* folder. Test the program.

Modify the program so that the crosshairs move only half the distance returned by the joystick.

SetVirtualJoystickVisible()

To modify a virtual joystick's visibility, use `SetVirtualJoystickVisible()` (see FIG-15.83).

FIG-15.83

SetVirtualJoystickVisible()

SetVirtualJoystickVisible ((id , iv))

where:

- id** is an integer value giving the ID of the joystick involved.
- iv** is an integer value (0 or 1) specifying whether the joystick is to be made visible (1) or invisible (0).

GetVirtualJoystickExists()

To check if a joystick with a specific ID currently exists, you can use the `GetVirtualJoystickExists()` statement (see FIG-15.84).

FIG-15.84

GetVirtualJoystickExists()

integer `GetVirtualJoystickExists` ((`id`))

where:

- id** is an integer value giving the ID to be checked.

The function returns 1 if a joystick of that ID exists, otherwise zero is returned.

DeleteVirtualJoystick()

When a virtual joystick is no longer required, it can be deleted using the `DeleteVirtualJoystick()` statement (see FIG-15.85).

FIG-15.85

DeleteVirtualJoystick()

`DeleteVirtualJoystick` ((`id`))

where:

- id** is an integer value giving the ID of the virtual joystick to be deleted.

NOTE: A maximum of four virtual joysticks can exist simultaneously within a project.

Physical Joysticks

Previously, we looked at the set of commands for creating and handling a virtual joystick, but AGK also has commands for using a real joystick. Of course, this would be inappropriate when your game is going to be played on a mobile device; on the other hand, if you have created your project specifically for a PC or Mac, then you may want to make use of a physical joystick rather than a virtual one.

But, even when you code assuming the presence of a physical joystick, AGK will create a virtual joystick replacement for that physical joystick if it is not connected when the game is executing.

SetJoystickScreenPosition()

Since there is no way of knowing in advance if your program is being run on a hardware setup that actually has a joystick, it is always wise to execute the `SetJoystickScreenPosition()` statement. If no physical joystick or keyboard (which can be used to emulate a joystick) is connected, this command will create a virtual joystick on the screen. However, if a real joystick or keyboard is available, this command will have no effect. The format for `SetJoystickScreenPosition()` is

FIG-15.86

given in FIG-15.86.

SetJoystickScreenPosition() **SetJoystickScreenPosition** ((x , y , rwidth)

where:

x,y are real values giving the coordinates of the virtual joystick.

rwidth is a real value giving the width of the joystick image.

The virtual joystick created will be assigned an ID of 1. If your program has already created a virtual joystick (using the **AddVirtualJoystick()** command) with the same ID, then that existing virtual joystick will be used.

If a virtual joystick appears, it will use the same default graphics as a virtual joystick created with the **AddVirtualJoystick()** statement.

Activity 15.29

Start a new project called *Joystick2*. Code *main.agc* as:

```
SetJoystickScreenPosition(50,50,20)
do
    Sync()
loop
```

Run the program on your PC without connecting a joystick. Does the virtual joystick appear on the screen?

Download the app to your tablet or smartphone. Does the virtual joystick appear?

Save your project.

GetJoystickX() and GetJoystickY()

The position of the joystick lever is read using the **GetJoystickX()** and **GetJoystickY()** statements. These statements have the format shown in FIG-15.87 and FIG-15.88.

FIG-15.87

GetJoystickX()

float **GetJoystickX** (())

FIG-15.88

GetJoystickY()

float **GetJoystickY** (())

The values returned by these statements will lie in the range -1 to 1.

If no physical joystick exists, but a physical keyboard is connected, then the keys *W* and *S* will be used to represent up and down movement respectively, while *A* and *D* are used for left and right movement. The *W* key will return a value in the range 0 to -1. The value returned increases the longer the key is held down, reaching -1 after the *W* key has been held down for about 0.5 seconds. The other keys react in a similar manner (S: 0 to 1; A: 0 to -1; D: 0 to 1).

SetJoystickDeadZone()

The `SetJoystickDeadZone()` statement sets the dead zone area for the joystick where the *x* and *y* values returned are zero, even though the stick may be shifted slightly from centre. The default setting for the dead zone is 0.15 in all directions.

FIG-15.89

This statement has the format shown in FIG-15.89.

SetJoystickDeadZone()

`SetJoystickDeadZone` (`fzone`)

where:

fzone is a real number giving the smallest absolute value detectable using the joystick. This should be in the range 0 to 1.

Activity 15.30

Modify *Joystick2* so that it controls the movement of the crosshairs (*crosshairs.png*) previously used in Activity 15.28.

Test and save your project.

SetButtonScreenPosition()

Most real joysticks have several buttons. These allow players to do things such as boost speed or fire a weapon.

If the keyboard is being used in place of a real joystick, then the keys *Space*, *E*, *R*, *Q*, and *Ctrl* are used in place of the joystick's buttons.

When using the joystick commands (as opposed to the virtual joystick commands) you can ensure that a set of buttons exists on the screen when the player has no physical joystick and no physical keyboard by using the `SetButtonScreenPosition()` statement. If the setup does include a real joystick or keyboard, this command has no effect. The statement's format is shown in FIG-15.90.

FIG-15.90

SetButtonScreenPosition()

`SetButtonScreenPosition` (`ibutton` , `x` , `y` , `rwidth`)

where:

ibutton is an integer number identifying the button. This must be in the range 1 to 5.

x,y are real values giving the coordinates of the virtual button.

rwidth is a real value giving the width of the button image.

GetButtonPressed()

To check if a button on the joystick has been pressed, use the `GetButtonPressed()` statement (see FIG-15.91).

FIG-15.91

GetButtonPressed()

integer `GetButtonPressed` (`ibutton`)

where:

ibutton is an integer number identifying the button to be checked. This must be in the range 1 to 5.

The function returns 1 at the moment the button is first pressed; at all other times zero is returned.

GetButtonReleased()

To check if a joystick has just been released, use the `GetButtonReleased()` statement (see FIG-15.92).

FIG-15.92

`GetButtonReleased()` integer `GetButtonReleased()` (`ibutton`)

where:

ibutton is an integer number identifying the button to be checked. This must be in the range 1 to 5.

The function returns 1 at the moment the button is first released, at all other times zero is returned.

GetButtonState()

While `GetButtonPressed()` returns 1 only for an instant as the joystick button is first pressed, and `GetButtonReleased()` returns 1 only at the instant the joystick button is released, `GetButtonState()` can be used to determine if a joystick button is currently being held down or is untouched. The statement has the format shown in FIG-15.93.

FIG-15.93

`GetButtonState()` integer `GetButtonState()` (`ibutton`)

where:

ibutton is an integer value (1 to 5) giving the joystick button to be tested.

The function returns 1 if the specified button is currently being pressed, otherwise zero is returned.

Summary

- AGK contains instructions to use both virtual and physical joysticks.
- A virtual joystick is constructed from two graphic elements known as the inner joystick and the outer joystick.
- The inner joystick represents the joystick's moveable arm; the outer joystick represents the fixed joystick casing.
- Use `AddVirtualJoystick()` to create a virtual joystick. This uses the default graphics.
- Use `SetVirtualJoystickPosition()` to reposition the virtual joystick on the screen.
- Use `SetVirtualJoystickSize()` to resize the virtual joystick.
- Use `SetVirtualJoystickAlpha()` to adjust the transparency of the virtual

joystick's image.

- Use `SetVirtualJoystickInner()` to use a different graphic for the inner part of the virtual joystick.
- Use `SetVirtualJoystickOuter()` to use a different graphic for the outer part of the virtual joystick.
- Use `GetVirtualJoystickX()` and `GetVirtualJoystickY()` to determine the offset from the centre of the joystick arm. Values returned lie in the range -1 to 1.
- A joystick's dead zone is the area (near the central position) in which the arm offset is assumed to be zero. By default, the dead zone operates for offsets whose absolute value is less than 0.15.
- Use `SetVirtualJoystickDeadZone()` to specify how far the joystick arm must be moved from its central position before a reading other than zero is registered.
- Use `SetVirtualJoystickActive()` to disable/enable the use of the virtual joystick.
- Use `SetVirtualJoystickVisible()` to make a joystick visible/invisible.
- Use `GetVirtualJoystickExists()` to check for the existence of a virtual joystick with a specified ID.
- Use `DeleteVirtualJoystick()` to delete a virtual joystick resource.
- Physical joysticks can be used when running the app on a device which allows for a joystick connection.
- Where no joystick is attached when the app is running, the *WSAD* keys are used in place of the joystick lever.
- Use `SetJoystickScreenPosition()` to specify where a virtual joystick should be positioned if no physical joystick or keyboard is connected.
- Use `GetJoystickX()` and `GetJoystickY()` to determine the offset from the centre of the joystick arm. The *WSAD* keys are used where only a keyboard is attached. Values returned lie in the range -1 to 1.
- Use `SetJoystickDeadZone()` to specify how far the joystick arm must be moved from its central position before a reading other than zero is registered.
- If a physical joystick has no buttons, the keys *Space*, *E*, *R*, *Q*, and *Ctrl* will be used as the first five buttons of the joystick.
- Where no joystick buttons are available and no physical keyboard is connected, then use `SetButtonScreenPosition()` to create substitute virtual buttons.
- Use `GetButtonPressed()` to check if a joystick button has just been pressed.
- Use `GetButtonReleased()` to check if a joystick button has just been released.
- Use `GetButtonState()` to determine if a specified joystick button is currently pressed.

Device Dependent Input

Introduction

Although the main aim of AGK is to create applications that can run on just about any platform, there are a few statements which only function on specific devices. For example, there are statements to handle an accelerometer (the hardware that detects the orientation of your phone or tablet) and to handle real keyboards, mice, or joysticks. While the accelerometer will not be available on a PC, the keyboard, mouse and joystick are unlikely to be available on a phone or tablet. By using these statements, you limit the platforms on which your app will run.

Accelerometer Statements

GetAccelerometerExists()

We can check if the device running your app contains an accelerometer using the `GetAccelerometerExists()` statement (see FIG-15.94).

FIG-15.94

GetAccelerometerExists()

integer `GetAccelerometerExists()` ()

The function returns 1 if the device running the app contains an accelerometer, otherwise zero is returned.

The program in FIG-15.95 displays the value returned by the `GetAccelerometerExists()` statement.

FIG-15.95

Checking for an
Accelerometer

```
rem *** Using the Accelerometer ***  
  
do  
    rem *** Check if accelerometer exists ***  
    r = GetAccelerometerExists()  
    rem *** Display result ***  
    Print(r)  
    Sync()  
loop
```

Activity 15.31

Start a new project called *Accelerometer* and implement the code given in FIG-15.95.

Run the program on your PC and phone/tablet. What result do you produce from each device?

Save your project.

GetDirectionAngle()

The `GetDirectionAngle()` returns the angle (0 to 360) at which the device running the app is being held relative to the inverted portrait orientation (see FIG-15.96).

FIG-15.96

Angle Detection

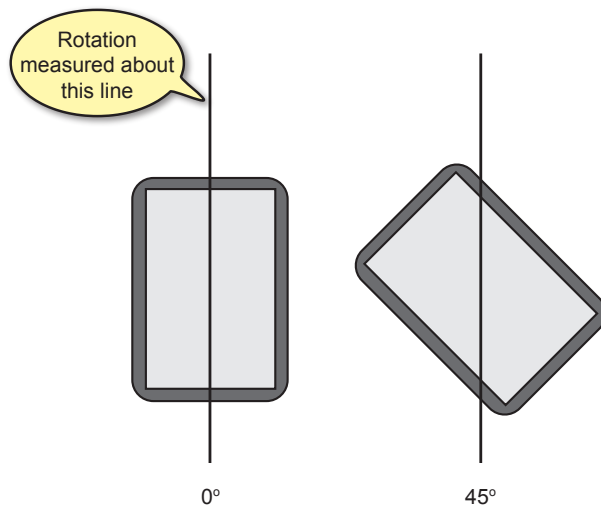


FIG-15.97

GetDirectionAngle()

The `GetDirectionAngle()` statement has the format shown in FIG-15.97.

float `GetDirectionAngle` ()

When used on a PC, you can emulate the basic rotation using the arrow keys, although this only gives results in 90° steps.

Activity 15.32

Modify *Accelerometer* so that it displays the angle at which your device is being held.

Run the program on your PC and use the arrow keys to emulate rotation.

Run the program on your phone/tablet. What orientation gives a reading of zero degrees?

Save your project.

GetDirectionX() and GetDirectionY()

The `GetDirectionX()` and `GetDirectionY()` statements return (x,y) coordinates based on the tilt and angle of the device. All values are in the range 0 to 1. On the PC, the arrow keys can be used to emulate the movement. Typical values are shown in FIG-15.98.

FIG-15.98

Device Coordinates

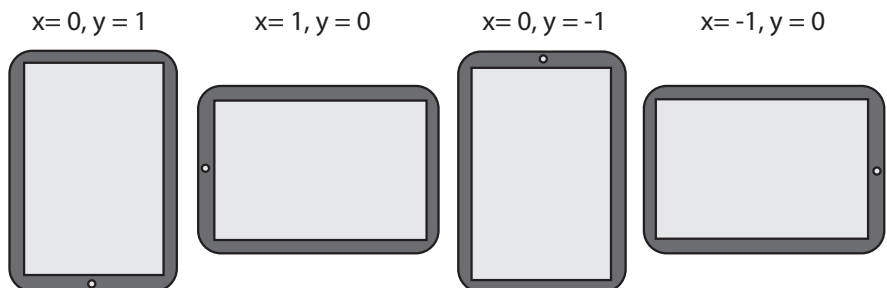


FIG-15.99

GetDirectionX()

GetDirectionY()

The statements' formats are shown in FIG-15.99.

float `GetDirectionX ()`float `GetDirectionY ()`**Activity 15.33**

Modify *Accelerometer* so that it displays the *x* and *y* coordinates of your device.

Test and save your project.

GetRawAccelX() GetRawAccelY() and GetRawAccelZ()

You can find the orientation in 3D space of any device with an accelerometer. The statements `GetRawAccelX()`, `GetRawAccelY()` and `GetRawAccelZ()` return values in the range -1 to 1 based on the orientation of your device. The format for each statement is given in FIG-15.100.

FIG-15.100

GetRawAccelX()

GetRawAccelY()

GetRawAccelZ()

float `GetRawAccelX ()`float `GetRawAccelY ()`float `GetRawAccelZ ()`**Activity 15.34**

Modify *Accelerometer* so that it displays the *x*, *y* and *z* coordinates of your device. Observe how moving the device relates to the values displayed

Save your project.

GetDirectionSpeed()

The `GetDirectionSpeed()` statement's name is perhaps a little confusing, since the output it produces is really a combination of the device's orientation and the speed at which it is being moved (shaken). The statement has the format shown in FIG-15.101.

FIG-15.101

GetDirectionSpeed()

float `GetDirectionSpeed ()`

The value returned normally lies between 0 and about 2.9.

Activity 15.35

Modify *Accelerometer* so that it displays the "speed" of your device.

Change the code so that the highest value returned is displayed.

Test and save your project.

Mouse Statements

If you are sure your app is going to be run on a device which uses a mouse, many mouse-specific commands are available. These are listed below.

GetMouseExists()

The `GetMouseExists()` statement returns 1 if a mouse is attached to the device running the app, otherwise zero is returned (see FIG-15.102).

FIG-15.102

GetMouseExists()

integer `GetMouseExists()` ()

GetRawMouseLeftPressed()

The instant the left mouse button is pressed, `GetRawMouseLeftPressed()` returns 1; at all other times zero is returned (see FIG-15.103).

FIG-15.103

GetRawMouseLeft
↳ Pressed()

integer `GetRawMouseLeftPressed()` ()

GetRawMouseLeftReleased()

The instant the left mouse button is released, `GetRawMouseLeftReleased()` returns 1; at all other times zero is returned (see FIG-15.104).

FIG-15.104

GetRawMouseLeft
↳ Released()

integer `GetRawMouseLeftReleased()` ()

GetRawMouseLeftState()

To discover if the left mouse button is currently being pressed, use `GetRawMouseLeftState()` which returns 1 when the button is being held down, and zero at all other times (see FIG-15.105).

FIG-15.105

GetRawMouseLeftState()

integer `GetRawMouseLeftState()` ()

GetRawMouseRightPressed()

The instant the right mouse button is pressed, `GetRawMouseRightPressed()` returns 1; at all other times zero is returned (see FIG-15.106).

FIG-15.106

GetRawMouseRight
↳ Pressed()

integer `GetRawMouseRightPressed()` ()

GetRawMouseRightReleased()

The instant the right mouse button is released, `GetRawMouseRightReleased()` returns 1; at all other times zero is returned (see FIG-15.107).

FIG-15.107

GetRawMouseRight
↳ Released()

integer `GetRawMouseRightReleased()` ()

GetRawMouseRightState()

To discover if the right mouse button is currently being pressed, use `GetRawMouseRightState()` which returns 1 when the button is being held down, and

FIG-15.108

zero at all other times (see FIG-15.108).

GetRawMouseRightState()

integer **GetRawMouseRightState** () ()

GetRawMouseX() and GetRawMouseY()

The coordinates of the mouse pointer can be found using the **GetRawMouseX()** and **GetRawMouseY()** statements (see FIG-15.109).

FIG-15.109

GetRawMouseX()

float **GetRawMouseX** () ()

GetRawMouseY()

float **GetRawMouseY** () ()

The program in FIG-15.110 makes use of various raw mouse statements, displaying the state of any mouse found.

FIG-15.110

Using Raw Mouse
Statements

```
rem *** Raw Mouse Test ***

rem *** Create text objects ***
CreateText(1,"")
CreateText(2,"")
SetTextPosition(2,0,8)
CreateText(3,"")
SetTextPosition(3,0,16)
do
  rem *** IF mouse exists THEN ***
  if GetMouseExists() = 1
    rem *** Display details ***
    SetTextString(1,"Left mouse button : "+
    ↳Str(GetRawMouseLeftState()))
    SetTextString(2,"Right mouse button : "+
    ↳Str(GetRawMouseRightState()))
    SetTextString(3,"Mouse coordinates : (" +
    ↳Str(GetRawMouseX(),2)+" , "+Str(GetRawMouseY(),2)+" ")
  else
    rem *** If no mouse, display message ***
    SetTextString(1,"No mouse")
  endif
  Sync()
loop
```

Activity 15.36

Start a new project called *RawMouse* and implement the code in FIG-15.110.

Test the program on your PC and tablet/phone. Save your project.

SetRawMouseVisible()

You can hide/show the mouse pointer (if one exists) using **SetRawMouseVisible()** (see FIG-15.111).

FIG-15.111

SetRawMouseVisible()

SetRawMouseVisible () **iop** ()

where

iop is an integer value (0: hide pointer or 1: show pointer).

Joystick Statements

We have already met statements which can handle real joysticks, but those would automatically substitute a virtual joystick if the real one is missing. However, a further set of joystick commands are available which only work when a physical joystick is present.

CompleteRawJoystickDetection()

It may take a few seconds to detect all of the joysticks attached to a device. The `CompleteRawJoystickDetection()` statement should be used to ensure that the detection process has been completed.

FIG-15.112

CompleteRawJoystick
↳ Detection()

The statement has the format shown in FIG-15.112.

CompleteRawJoystickDetection ()

The joysticks must already be connected to the device before this statement is executed. They will not be detected by the system if added later.

GetRawJoystickExists()

Up to four joysticks can be detected by AGK, each being giving an ID from 1 to 4. To detect if a joystick of a given ID is present we can use `GetRawJoystickExists()` (see FIG-15.113).

FIG-15.113

GetRawJoystickExists()

integer GetRawJoystickExists (id)

where

id is an integer value (1 to 4) giving the ID to be tested.

GetRawJoystickButtonPressed()

The instant a joystick button is pressed, `GetRawJoystickButtonPressed()` returns 1; at all other times zero is returned (see FIG-15.114).

FIG-15.114

GetRawJoystickButton
↳ Pressed()

integer GetRawJoystickButtonPressed (id , ibut)

where

id is an integer value (1 to 4) giving the ID of the joystick.

ibut is an integer value (1 to 32) giving the number of the button to be checked.

GetRawJoystickButtonReleased()

The instant a joystick button is released, `GetRawJoystickButtonReleased()` returns 1; at all other times zero is returned (see FIG-15.115).

FIG-15.115

GetRawJoystickButton
↳ Released()

integer GetRawJoystickButtonReleased (id , ibut)

where

id is an integer value (1 to 4) giving the ID of the joystick.

ibut is an integer value (1 to 32) giving the number of the button to be checked.

GetRawJoystickButtonState()

To discover if a joystick button is currently being pressed, use `GetRawJoystickButtonState()` which returns 1 when the button specified is being held down, and zero at all other times (see FIG-15.116).

FIG-15.116

`GetRawJoystickButtonState()`

integer `GetRawJoystickButtonState` ((id , ibut))

where

id is an integer value (1 to 4) giving the ID of the joystick.

ibut is an integer value (1 to 32) giving the number of the button to be checked.

GetRawJoystickX(), GetRawJoystickY() and GetRawJoystickZ()

To detect how far a joystick has moved off-centre in the *x*, *y* and (where appropriate) the *z* directions, we use the statements `GetRawJoystickX()`, `GetRawJoystickY()`, and `GetRawJoystickZ()` (see FIG-15.117).

FIG-15.117

`GetRawJoystickX()`

`GetRawJoystickY()`

`GetRawJoystickZ()`

float `GetRawJoystickX` ((id))

float `GetRawJoystickY` ((id))

float `GetRawJoystickZ` ((id))

where

id is an integer value (1 to 4) giving the ID of the joystick.

The values returned lie in the range -1 to 1.

Most joysticks will only return *x* and *y* readings.

GetRawJoystickRX(), GetRawJoystickRY() and GetRawJoystickRZ()

The angle to which the joystick has been rotated about the *x*, *y* and *z* axes can be determined using `GetRawJoystickRX()`, `GetRawJoystickRY()` and `GetRawJoystickRZ()` (see FIG-19.118).

FIG-15.118

`GetRawJoystickRX()`

`GetRawJoystickRY()`

`GetRawJoystickRZ()`

float `GetRawJoystickRX` ((id))

float `GetRawJoystickRY` ((id))

float `GetRawJoystickRZ` ((id))

where

id is an integer value (1 to 4) giving the ID of the joystick.

The values returned lie in the range -1 to 1.

Most joysticks will detect rotation about the z-axis only.

SetRawJoystickDeadZone()

The dead zone is the area near the centred-position of the joystick where the joystick *x* and *y* values are returned as zero. By default this is set to 0.15; as the reading from `GetRawJoystickX()` (or `GetRawJoystickY()`) gets close to ± 0.15 and the joystick heads towards the central position, the value returned will suddenly jump to 0.

You can change the value reached before zero is returned by these functions using the `SetRawJoystickDeadZone()` statement (see FIG-15.119).

FIG-15.119

SetRawJoystickDead
Zone()

`SetRawJoystickDeadZone ((fv))`

where

fv is a real value specifying the absolute minimum value before zero is returned when reading the joystick's *x* and *y* positions.

The program in FIG-15.120 gives a live readout for a joystick 1 and button 1.

FIG-15.120

Using Raw Joystick
Statements

```
rem *** Raw Joystick Test ***

rem *** Create text objects ***
CreateText(1,"")
CreateText(2,"")
SetTextPosition(2,0,8)
CreateText(3,"")
SetTextPosition(3,0,16)

rem *** Test for joysticks ***
CompleteRawJoystickDetection()

do
  rem *** IF joystick exists THEN ***
  if GetRawJoystickExists(1) = 1
    rem *** Display details ***
    SetTextString(1,"Joystick button 1 : "+
    ⤵Str(GetRawJoystickButtonState(1,1))
    SetTextString(2,"Joystick offsets : "+
    ⤵Str(GetRawJoystickX(1),2)+" "+
    ⤵Str(GetRawJoystickY(1),2)+" "+Str(GetRawJoystickZ(1),2))
    SetTextString(3,"Joystick Rotation : "+
    ⤵Str(GetRawJoystickRX(1),2)+" "+Str(GetRawJoystickRY(1),
    ⤵2)+" "+Str(GetRawJoystickRZ(1),2))
  else
    rem *** If no joystick, display message ***
    SetTextString(1,"No joystick")
  endif
  Sync()
loop
```

Activity 15.37

Start a new project called *RawJoystick* and implement the code in FIG-15.120.

Connect a joystick to your PC and test the program. Save your project.

Keyboard Statements

Where a keyboard is attached, the following commands may be used.

GetKeyboardExists()

To check that a keyboard is connected, use the `GetKeyboardExists()` statement (see FIG-15.121).

FIG-15.121

GetKeyboardExists() integer `GetKeyboardExists()`

The function returns 1 if a keyboard is connected, otherwise zero is returned.

GetRawKeyPressed()

You can check if a key has just been pressed using the `GetRawKeyPressed()` statement (see FIG-15.122).

FIG-15.122

GetRawKeyPressed() integer `GetRawKeyPressed(iascii)`

where

iascii is the integer ASCII value of the character whose key is to be checked.

The function returns 1 the instant the specified key is pressed; at all other times zero is returned.

For example, the line

```
Print (GetRawKeyPressed (65) )
```

would display zero until the “A” key is pressed. At that point the output would change to 1 - but only for a fraction of a second - before returning to zero again. The value 1 is returned when either the uppercase or lowercase version of the character is entered.

GetRawKeyReleased()

The moment a specified key is released `GetRawKeyReleased()` returns 1, at all other times zero is returned. The statement has the format shown in FIG-15.123.

FIG-15.123

GetRawKeyReleased() integer `GetRawKeyReleased(iascii)`

where

iascii is the integer ASCII value of the character whose key is to be checked.

GetRawKeyState()

When we need to know if a key is currently being pressed down or is untouched, we can use the `GetRawKeyState()` statement (see FIG-15.124).

FIG-15.124

`GetRawKeyState()`

integer `GetRawKeyState` (`iascii`)

where

iascii is the integer ASCII value of the character whose key is to be checked.

The function returns 1 when the specified key is being held down; zero when the key is untouched.

GetRawLastKey()

The ASCII code of the last key to be pressed can be discovered using the `GetRawLastKey()` statement (see FIG-15.125).

FIG-15.125

`GetRawLastKey()`

integer `GetRawLastKey` ()

Alphabetic keys always return the ASCII value of the uppercase character; the top row of keys returns the ASCII for the numeric digit on the key. The program in FIG-15.126 displays the state of the 'A' and also the last key to be pressed.

FIG-15.126

Using Raw Keyboard
Statements

```
rem *** Raw Keyboard Test ***

rem *** Create text objects ***
CreateText(1,"")
CreateText(2,"")
SetTextPosition(2,0,8)
do
    rem *** IF keyboard exists THEN ***
    if GetKeyboardExists() = 1
        rem *** Display details ***
        SetTextString(1,"'A' key state      : "+
        ↳Str(GetRawKeyState(65)))
        SetTextString(2,"Last key pressed  : "+
        ↳Chr(GetRawLastKey()))
    else
        rem *** If no keyboard, display message ***
        SetTextString(1,"No keyboard")
    endif
    Sync()
    Sleep(50)
loop
```

Activity 15.38

Start a new project called *RawKeyboard* and implement the code in FIG-15.126.

Check the output when the 'A' key is held down. What output is produced for 'Last key pressed : ' when a non-alphanumeric key is pressed? Save your project.

- Use `GetDirectionSpeed()` to read a value based on the device's angle and movement.
- Use `GetMouseExists()` to check if a mouse is attached to the device.
- Use `GetRawMouseLeftPressed()` to detect the instant the left mouse button is pressed.
- Use `GetRawMouseLeftReleased()` to detect the instant the left mouse button is released.
- Use `GetRawMouseLeftState()` to detect the current state of the left mouse button.
- Use `GetRawMouseRightPressed()` to detect the instant the right mouse button is pressed.
- Use `GetRawMouseRightReleased()` to detect the instant the right mouse button is released.
- Use `GetRawMouseRightState()` to detect the current state of the right mouse button.
- Use `GetRawMouseX()` and `GetRawMouseY()` to determine the position of the mouse pointer.
- Use `SetRawMouseVisible()` to make the mouse pointer visible/invisible.
- Use `CompleteRawJoystickDetection()` to ensure a device has finished checking for joysticks.
- Use `GetRawJoystickExists()` to check if a joystick of a specified ID exists.
- Use `GetRawJoystickButtonPressed()` to detect the instant a specific button on a stated joystick is pressed.
- Use `GetRawJoystickButtonReleased()` to detect the instant a specific button on a stated joystick is released.
- Use `GetRawJoystickButtonState()` to detect the current state of a specific button on a stated joystick.
- Use `GetRawJoystickX()`, `GetRawJoystickY()` and `GetRawJoystickZ()` to discover the position of a specified joystick. Usually only *x* and *y* are available.
- Use `GetRawJoystickRX()`, `GetRawJoystickRY()` and `GetRawJoystickRZ()` to discover the rotation of a specified joystick. Usually only *z* is available.
- Use `SetRawJoystickDeadZone()` to set the dead zone of a specific joystick.
- Use `GetKeyboardExists()` to check if a keyboard is currently connected.
- Use `GetRawKeyPressed()` to detect the instant a specified key is pressed.
- Use `GetRawKeyReleased()` to detect the instant a specified key is released.
- Use `GetRawKeyState()` to discover the current state of a specified key.
- Use `GetRawLastKey()` to discover the ASCII code of the last key pressed.
- Use `GetDeviceName()` to retrieve details of the OS (and perhaps hardware) which is running your app.

Solutions

Activity 15.1

Code for *UsingVirtualButtons*:

```
rem *** Using Virtual Buttons ***

rem *** Create button ***
AddVirtualButton(1,50,50,10)
do
    Sync()
loop
```

When the button is selected a different image is displayed to give the impression of a pressed button.

Activity 15.2

Modified code for *UsingVirtualButtons*:

```
rem *** Using Virtual Buttons ***

rem *** Create button ***
AddVirtualButton(1,50,50,10)
rem *** Add text to button ***
SetVirtualButtonText(1,"Yes")
do
    Sync()
loop
```

Activity 15.3

Modified code for *UsingVirtualButtons*:

```
rem *** Using Virtual Buttons ***

rem *** Create button ***
AddVirtualButton(1,50,50,10)
rem *** Add text to button ***
SetVirtualButtonText(1,"Yes")
rem *** Colour the button yellow ***
SetVirtualButtonColor(1,255,255,0)
do
    Sync()
loop
```

Activity 15.4

Modified code for *UsingVirtualButtons*:

```
rem *** Using Virtual Buttons ***

rem *** Create button ***
AddVirtualButton(1,50,50,10)
rem *** Add text to button ***
SetVirtualButtonText(1,"Yes")
rem *** Colour the button yellow ***
SetVirtualButtonColor(1,255,255,0)
rem *** Make button translucent ***
SetVirtualButtonAlpha(1,126)
do
    Sync()
loop
```

A translucent yellow button on a black background makes the button appear dull yellow.

Activity 15.5

Modified code for *UsingVirtualButtons*:

```
rem *** Using Virtual Buttons ***

rem *** Create button ***
AddVirtualButton(1,50,50,10)
rem *** Add text to button ***
SetVirtualButtonText(1,"Yes")
rem *** Colour the button yellow ***
SetVirtualButtonColor(1,255,255,0)
rem *** Make button translucent ***
SetVirtualButtonAlpha(1,126)
```

```
rem *** record start time ***
time = GetSeconds()
do
    rem *** After 5 seconds, inactivate button ***
    if GetSeconds() - time = 5
        SetVirtualButtonActive(1,0)
    endif
    Sync()
loop
```

Activity 15.6

No solution required.

Activity 15.7

Modified code for *VB2*:

```
rem *** Using Virtual Buttons 2 ***
rem *** Load images used ***
LoadImage(1,"AUp.png",0)
LoadImage(2,"ADown.png",0)
rem *** Create button ***
AddVirtualButton(1,50,90,10)
rem *** Add images to button ***
SetVirtualButtonImageUp(1,1)
SetVirtualButtonImageDown(1,2)
rem *** Create text object ***
text$ = ""
CreateText(1,"")
SetTextPosition(1,40,50)
do
    rem *** IF key presses, add an "a" ***
    if GetVirtualButtonPressed(1)=1
        text$=text$+"a"
        SetTextString(1,text$)
    endif
    Sync()
loop
```

Activity 15.8

No solution required.

Activity 15.9

No solution required.

Activity 15.10

Modified code for *KeyboardInput*:

```
StartTextInput()
do
    if GetTextInputCompleted() = 1
        CreateText(1,"Completed")
    endif
    Sync()
loop
```

Activity 15.11

Modified code for *KeyboardInput*:

```
StartTextInput()
do
    if GetTextInputCompleted() = 1
        CreateText(1,GetTextInput())
    endif
    Sync()
loop
```

Activity 15.12

Modified code for *KeyboardInput*:

```
StartTextInput()
do
    if GetTextInputCompleted() = 1
        if GetTextInputCancelled() = 1
            CreateText(1,"User cancelled")
        else
            CreateText(1,GetTextInput())
        endif
    endif
```

```

endif
Sync()
loop

```

Activity 15.13

Modified code for *KeyboardInput*:

```

rem *** Record current time ***
time = GetSeconds()
StartTextInput()
do
    if GetTextInputCompleted() = 1
        if GetTextInputCancelled() = 1
            CreateText(1,"User cancelled")
        else
            CreateText(1,GetTextInput())
        endif
    endif
    rem *** If 8 seconds past ***
    if GetSeconds()-time = 8
        rem *** Stop text input ***
        StopTextInput()
    endif
    Sync()
loop

```

Activity 15.14

No solution required.

Activity 15.15

No more than 15 characters are accepted from the keyboard.

Activity 15.16

Code for *EditBox01*:

```

rem *** Using Edit boxes ***

rem *** Create the edit box ***
CreateEditBox(1)
SetEditBoxSize(1,70,5)
SetEditBoxPosition(1,10,20)

rem *** Set text size ***
SetEditBoxTextSize(1,3,5)

rem *** Assign focus to the edit box ***
SetEditBoxFocus(1,1)
do
    rem *** If the box loses focus, display contents ***
    if GetEditBoxChanged(1) = 1
        Print("Text entered was : ")
        ⌘+GetEditBoxText(1)
    endif
    Sync()
loop

```

To remove the 15 character restriction, the `SetEditBoxMaxChars()` statement has been removed.

When the edit box is filled, the text it contains will scroll sideways to allow more characters to be entered.

Activity 15.17

Modified code for *EditBox01*:

```

rem *** Using Edit boxes ***

rem *** Create the edit box ***
CreateEditBox(1)
SetEditBoxSize(1,70,5)
SetEditBoxPosition(1,10,20)

rem *** Create second edit box ***
CreateEditBox(2)
SetEditBoxSize(2,70,20)
SetEditBoxPosition(2,10,30)
rem *** Max 7 lines ***
SetEditBoxMultiLine(2,1)

```

```
SetEditBoxMaxLines(2,7)
```

```

rem *** Set text size ***
SetEditBoxTextSize(1,2,5)
SetEditBoxTextSize(2,2,5)
rem *** Assign focus to first edit box ***
SetEditBoxFocus(1,1)
do
    rem *** If first box loses focus, move to second ***
    if GetEditBoxChanged(1) = 1
        SetEditBoxFocus(2,1)
    endif
    Sync()
loop

```

Activity 15.18

Modified code for *EditBox01*:

```

rem *** Using Edit boxes ***

rem *** Initial text in edit boxes ***
dim initialtext[2] as string=["","Name",
⌘"Postal Address"]

rem *** Create the name edit box ***
CreateEditBox(1)
SetEditBoxSize(1,70,5)
SetEditBoxPosition(1,10,20)
rem *** Set text size ***
SetEditBoxTextSize(1,2,5)

rem *** Set initial text ***
SetEditBoxText(1,initialtext[1])

rem *** Create address edit box ***
CreateEditBox(2)
SetEditBoxSize(2,70,20)
SetEditBoxPosition(2,10,30)
rem *** Max 7 lines ***
SetEditBoxMultiLine(2,1)
SetEditBoxMaxLines(2,7)
rem *** Set text size ***
SetEditBoxTextSize(2,2,5)

rem *** Set initial text ***
SetEditBoxText(2,initialtext[2])

do
    rem *** Get ID of current edit box ***
    id = GetCurrentEditBox()
    rem *** If an edit box is in focus ***
    if id <> 0
        rem *** If it contains its initial text ***
        if GetEditBoxText(id) = initialtext[id]
            rem *** Remove it ***
            SetEditBoxText(id,"")
        endif
        rem *** Reinstate descriptor in other box if empty ***
        if GetEditBoxChanged(3-id)= 1 and
            ⌘GetEditBoxText(3-id)=""
            SetEditBoxText(3-id,initialtext[3-id])
        endif
    endif
    Sync()
loop

```

Since the edit boxes have ID's 1 and 2, the unfocused box's ID can always be calculated as 3 - ID of focused box.

Activity 15.19

No solution required.

Activity 15.20

Modified code for *VirtualJoystick*:

```

rem *** Using a virtual joystick ***
rem *** Add joystick ***
SetClearColor(255,255,0)
AddVirtualJoystick(1,50,50,30)
rem *** Reposition joystick ***
SetVirtualJoystickPosition(1,80,80)
do

```

```

Sync()
loop

```

Activity 15.21

Modified code for *VirtualJoystick*:

```

rem *** Using a virtual joystick ***
rem *** Add joystick ***
SetClearColor(255,255,0)
AddVirtualJoystick(1,50,50,30)
rem *** Reposition joystick ***
SetVirtualJoystickPosition(1,80,80)
rem *** Resize joystick ***
SetVirtualJoystickSize(1,20)
do
    Sync()
loop

```

Activity 15.22

Modified code for *VirtualJoystick*:

```

rem *** Using a virtual joystick ***
rem *** Add joystick ***
SetClearColor(255,255,0)
AddVirtualJoystick(1,50,50,30)
rem *** Reposition joystick ***
SetVirtualJoystickPosition(1,80,80)
rem *** Resize joystick ***
SetVirtualJoystickSize(1,20)
rem *** Make joystick opaque ***
SetVirtualJoystickAlpha(1,255,255)
do
    Sync()
loop

```

Activity 15.23

Modified code for *VirtualJoystick*:

```

rem *** Using a virtual joystick ***
rem *** Load images required ***
LoadImage(1,"JoystickOuter.png",0)
LoadImage(2,"JoystickInner.png",0)
rem *** Add joystick ***
SetClearColor(255,255,0)
AddVirtualJoystick(1,50,50,30)
rem *** Change joystick images ***
SetVirtualJoystickImageOuter(1,1)
SetVirtualJoystickImageInner(1,2)
rem *** Reposition joystick ***
SetVirtualJoystickPosition(1,80,80)
rem *** Resize joystick ***
SetVirtualJoystickSize(1,20)
rem *** Make joystick opaque ***
SetVirtualJoystickAlpha(1,255,255)
do
    Sync()
loop

```

Activity 15.24

Modified code for *VirtualJoystick*:

```

rem *** Using a virtual joystick ***
rem *** Load images required ***
LoadImage(1,"JoystickOuter.png",0)
LoadImage(2,"JoystickInner.png",0)
rem *** Create Text objects ***
CreateText(1,"X: ")
SetTextColor(1,0,0,255)
CreateText(2,"Y: ")
SetTextColor(2,0,0,255)
rem *** Position text ***
SetTextPosition(1,5,5)
SetTextPosition(2,5,10)
rem *** Add joystick ***
SetClearColor(255,255,0)
AddVirtualJoystick(1,50,50,30)
rem *** Change joystick images ***
SetVirtualJoystickImageOuter(1,1)
SetVirtualJoystickImageInner(1,2)
rem *** Reposition joystick ***
SetVirtualJoystickPosition(1,80,80)
rem *** Resize joystick ***

```

```

SetVirtualJoystickSize(1,20)
rem *** Make joystick opaque ***
SetVirtualJoystickAlpha(1,255,255)
do
    rem *** Update display ***
    SetTextString(1,"X: "+Str(GetVirtualJoystickX(1)))
    SetTextString(2,"Y: "+Str(GetVirtualJoystickY(1)))
    Sync()
loop

```

Activity 15.25

The lowest absolute value in both the x and y directions is 0.1667.

Activity 15.26

To adjust the dead zone, add the lines

```

rem *** Adjust dead zone to 0.08 along both axes ***
SetVirtualJoystickDeadZone(0.08)

```

immediately after

```

AddVirtualJoystick(1,50,50,30)

```

Activity 15.27

Modified code for *VirtualJoystick*:

```

rem *** Using a virtual joystick ***
rem *** Load images required ***
LoadImage(1,"JoystickOuter.png")
LoadImage(2,"JoystickInner.png")
rem *** Create Text objects ***
CreateText(1,"X: ")
SetTextColor(1,0,0,255)
CreateText(2,"Y: ")
SetTextColor(2,0,0,255)
rem *** Position text ***
SetTextPosition(1,5,5)
SetTextPosition(2,5,10)
rem *** Add joystick ***
SetClearColor(255,255,0)
AddVirtualJoystick(1,50,50,30)
rem *** Adjust dead zone to 0.08 along both axes ***
SetVirtualJoystickDeadZone(0.08)
rem *** Change joystick images ***
SetVirtualJoystickImageOuter(1,1)
SetVirtualJoystickImageInner(1,2)
rem *** Reposition joystick ***
SetVirtualJoystickPosition(1,80,80)
rem *** Resize joystick ***
SetVirtualJoystickSize(1,20)
rem *** Make joystick opaque ***
SetVirtualJoystickAlpha(1,255,255)
rem *** set joystick state ***
joystickactive = 1
rem *** start timer ***
time = Timer()
do
    if Timer() - time >= 5 and joystickactive = 1
        SetVirtualJoystickActive(1,0)
        joystickactive = 0
    elseif Timer() - time >= 8 and joystickactive = 0
        SetVirtualJoystickActive(1,1)
        joystickactive = 2
    endif
    rem *** Update display ***
    SetTextString(1,"X: "+Str(GetVirtualJoystickX(1)))
    SetTextString(2,"Y: "+Str(GetVirtualJoystickY(1)))
    Sync()
loop

```

Activity 15.28

To halve the effect of the joystick on the movement of the crosshairs, all that is required is that the line

```

SetSpritePosition(1,GetSpriteX(1)+
    ↪GetVirtualJoystickX(1), GetSpriteY(1)+
    ↪GetVirtualJoystickY(1))

```

be changed to

```
SetSpritePosition(1, GetSpriteX(1)+
↳GetVirtualJoystickX(1)/2, GetSpriteY(1)+
↳GetVirtualJoystickY(1)/2)
```

Activity 15.29

Since you have a keyboard connected to the PC, that will be used as a substitute for the joystick and so no virtual joystick will appear.

When you run the app on your tablet, a virtual joystick will appear at the centre of the screen.

Activity 15.30

Modified code for *Joystick2*:

```
rem *** Control crosshairs using joystick ***

rem *** Set screen to grey ***
SetClearColor(120,120,120)
Sync()
rem *** Load image ***
LoadImage(1,"Crosshairs.png")
rem *** Create and position crosshairs sprite ***
CreateSprite(1,1)
SetSpriteSize(1,10,-1)
SetSpritePosition(1,45,45)
rem *** Allow for virtual joystick if no physical
rem *** joystick or keyboard available ***
SetJoystickScreenPosition(80,80,20)
do
    rem *** Update crosshairs' position ***
    SetSpritePosition(1, GetSpriteX(1)+
↳GetJoystickX()/2, GetSpriteY(1)+GetJoystickY()/2)
    Sync()
loop
```

Activity 15.31

A PC will display the value zero since no accelerometer will be present. Most mobile devices should display 1.

Activity 15.32

Modified code for *Accelerometer*:

```
rem *** Using the Accelerometer ***
do
    rem *** Get angle***
    r# = GetDirectionAngle()
    rem *** Display result ***
    Print(r#)
    Sync()
loop
```

On the iPad and ASUS Transformer, the device must be in inverted portrait mode to return a reading of zero.

Activity 15.33

Modified code for *Accelerometer*:

```
rem *** Using the Accelerometer ***
do
    rem *** Get coordinates ***
    x# = GetDirectionX()
    y# = GetDirectionY()
    rem *** Display result ***
    Print("X: "+Str(x#,2)+" "+Str(y#,2))
    Sync()
loop
```

Activity 15.34

Modified code for *Accelerometer*:

```
rem *** Using the Accelerometer ***
do
    rem *** Get coordinates ***
    x# = GetRawAccelX()
    y# = GetRawAccelY()
    z# = GetRawAccelZ()
    rem *** Display result ***
```

```
Print("X: "+Str(x#,2)+" Y: "+Str(y#,2)+" Z: "
↳+Str(z#,2))
Sync()
loop
```

Activity 15.35

Modified code for *Accelerometer*:

```
rem *** Using the Accelerometer ***
do
    rem *** Get speed ***
    speed# = GetDirectionSpeed()
    rem *** Display results ***
    Print("Speed : "+Str(speed#,2))
    Sync()
loop
```

Modified code for *Accelerometer* (highest speed):

```
rem *** Using the Accelerometer ***
highest# = 0
do
    rem *** Get speed ***
    speed# = GetDirectionSpeed()
    if speed# > highest#
        highest# = speed#
    endif
    rem *** Display result ***
    Print("Highest speed : "+Str(highest#,2))
    Sync()
loop
```

Activity 15.36

No solution required.

Activity 15.37

No solution required.

Activity 15.38

Some non-alphanumeric keys display no value (this is because the ASCII code returned is not in the range (33 to 126); others will return characters that do not match the key pressed.

Activity 15.39

No solution required.

Images

In this Chapter:

- ☐ Determining the Dimensions of an Image
- ☐ Atlas Texture Images
- ☐ The ImageJoiner Utility
- ☐ Loading Proportional Fonts
- ☐ Capturing an Image from the Screen
- ☐ Selecting Part of an Image
- ☐ User Image Selection
- ☐ Capturing an Image from the Camera
- ☐ Using Image Filters and Mipmaps

Introduction

In Chapter 7 we covered several commands from various resource types such as images and sprites. In this chapter we are going to cover the remaining image and sprite commands as well as look at a few more techniques which make use of these resources.

Review

We'll start by listing the image commands covered back in Chapter 7. These were:

<code>LoadImage (id, sfile)</code>	<i>id</i> is the ID to be assigned to the image. <i>sfile</i> is the name of the file containing the image.
<code>LoadImage (id, sfile, ialpha)</code>	<i>id</i> is the ID to be assigned to the image. <i>sfile</i> is the name of the file containing the image. <i>ialpha</i> is 0 to use image transparency and 1 to make black pixels invisible.
<code>int LoadImage (sfile)</code>	<i>sfile</i> is the name of the image file. The ID assigned is returned by the function.
<code>int LoadImage (sfile, ialpha)</code>	<i>sfile</i> is the name of the file containing the image. <i>ialpha</i> is 0 to use image transparency and 1 to make black pixels invisible. The function returns the ID assigned.
<code>DeleteImage (id)</code>	<i>id</i> is the ID of the image to be deleted.

The term *int* is used here as a shortened form of the word *integer*.

Further Image Statements

GetImageExists()

Although an image may be successfully loaded, we cannot always be sure that it still exists at some point later in the program, since it may have been deleted using the `DeleteImage ()` statement. To check that an image of a specified ID still exists, the `GetImageExists ()` statement can be used. This statement has the format shown in FIG-16.1.

FIG-16.1

GetImageExists()

integer `GetImageExists (id)`

where:

id is an integer value giving the ID that was assigned to the image by the `LoadImage ()` statement.

The statement returns 1 if the image was successfully loaded, otherwise zero is returned.

GetWidth()

We can discover the width of a loaded image using the `GetWidth()` statement which has the format shown in FIG-16.2.

FIG-16.2

GetWidth()

integer `GetWidth` (`id`)

where:

id is an integer value giving the ID that was assigned to the image.

The statement returns the width of the image in pixels.

GetHeight()

The height of an image can be determined using the `GetHeight()` statement which has the format shown in FIG-16.3.

FIG-16.3

GetHeight()

integer `GetHeight` (`id`)

where:

id is an integer value giving the ID that was assigned to the image.

The function returns the height of the image in pixels.

Activity 16.1

Start a new project called *ImageProperties*. Compile the default code and copy the file *HandsOnAGK/Chapter16/Size.png* into the project's *media* folder.

Code *main.agc* so that the image *Size.png* is loaded and this is followed by a 50% chance of the image being deleted.

If the image still exists, display the message *Image found* as well as the dimensions of the image. Have the code display *Image not found* if the image no longer exists.

Save your project.

LoadSubImage()

Some programs may make use of a very large number of images. When this is true, one way of simplifying the situation is to combine all of these images into one large composite image and then have the program split the large image into the original smaller ones from which it was constructed. This type of image is known as an **atlas texture image** and the images held within it as **subimages**.

The atlas image itself can be created using a paint package such as Photoshop where the separate images can be pasted onto a single canvas.

In addition to the atlas texture image, a separate text file is required. This text file contains the details of the subimages within the atlas image.

For each subimage in the atlas file, the following details are included:

- name assigned to the subimage
- the coordinates of the top left corner of the subimage (in pixels)
- the width and height of the subimage (in pixels)

The text file contains one line of data for each subimage and the data elements within a line are separated by colons.

The text file's name must be of a specific format, starting with the name of the atlas file followed by *subimages.txt*.

The atlas image shown in FIG-16.4 is made up of three separate images and is called *CompositelImage.png*.

FIG-16.4

An Atlas Texture Image



The accompanying text file must therefore be called:

`CompositelImage subimages.txt`

Note the required space between the two parts of the filename.

The contents of the text file are:

```
orchid.png:0:0:523:800
bunny.png:523:0:255:589
rock.png:523:589:255:211
```

The first line tells us that the atlas file contains a subimage which is to be called *orchid.png* and that this subimage's top left corner is at position (0,0) within the atlas image. The subimage is 523 pixels wide and 800 pixels high.

From this example we can see the required format for each line of the text file (see FIG-16.5).

FIG-16.5

Text File Line Format

filename : x : y : width : height

where:

- filename** is a string giving the filename to be assigned to the subimage.
- x,y** are integer values giving the coordinates (in pixels) of the top left-corner of the subimage.
- width** is an integer value giving the width of the subimage in pixels.
- height** is an integer value giving the height of the subimage in pixels.

The contents of the text file should be created using a simple text editor such as Microsoft's Notepad. Using a standard word processor to create the file will result in unwanted (but hidden) characters being added to your text.

The text file containing this information must be in the project's *media* folder and conform to the required naming rules (see FIG-16.6).

FIG-16.6

Text File Name Format

atlas image filename subimage.txt

where:

- atlas image filename** is a string giving the name of the file containing the atlas texture image. The atlas image file extension is not included in this string. For example, if the atlas image file is called *ImageCollection.png*, this string would be *ImageCollection*

Once the atlas image and the corresponding text file have been created and copied into the project's *media* folder, the atlas image is loaded using a standard `LoadImage ()` statement.

We can then extract the subimages using AGK BASIC's `LoadSubImage ()` statement. This statement has the format shown in FIG-16.7.

FIG-16.7

LoadSubImage()

Version 1

LoadSubImage (id , IdAtlas , sfile)

Version 2

integer LoadSubImage (IdAtlas , sfile)

where:


- id** is an integer value giving the ID to be assigned to the subimage being extracted from the atlas image.
- idAtlas** is an integer value giving the ID assigned to the atlas image when it was loaded.
- sfile** is a string giving the name of the file to be extracted from the atlas file. This name must appear within the subimages text file

which accompanies the atlas file.

FIG-16.8 shows the main steps involved in extracting the subimages from *CompositeImage.png*.

FIG-16.8

Loading the SubImages

After creating the atlas image and corresponding text file, we start a new AGK project and copy the two files to the project's <i>media</i> folder.	In the program code we start by loading the atlas image file using a standard <code>LoadImage()</code> statement.
<div>CompositelImage.png</div> <div></div> <div>CompositelImage subimages.txt</div> <div>orchid.png:0:0:523:800 bunny.png:800:0:255:589 rock.png:800:589:255:211</div>	atlasimageID = LoadImage("CompositelImage.png")
Now we extract the subimages from the atlas file using <code>LoadSubImage()</code> statements.	The subimages can then be treated as standard images and assigned to sprites in order to display them on the screen.
LoadSubImage(1,atlasimageId,"orchid.png") LoadSubImage(2,atlasimageId,"bunny.png") LoadSubImage(3,atlasimageId,"rock.png")	CreateSprite(1,1) CreateSprite(2,2) CreateSprite(3,3)

A complete program for loading and displaying the images is shown in FIG-16.9.

FIG-16.9

Using an Atlas Texture Image

```
rem *** Load atlas image ***
atlasimageID = LoadImage("CompositeImage.png")
rem *** Load subimages ***
LoadSubImage(1,atlasimageId,"orchid.png")
LoadSubImage(2,atlasimageId,"bunny.png")
LoadSubImage(3,atlasimageId,"rock.png")
rem *** Assign subimages to sprites ***
CreateSprite(1,1)
CreateSprite(2,2)
CreateSprite(3,3)
rem *** Resize and position sprites ***
SetSpriteSize(1,62,-1)
SetSpriteSize(2,25,-1)
SetSpriteSize(3,25,-1)
SetSpritePosition(2,70,0)
SetSpritePosition(3,70,70)
Sync()
do
loop
```

Activity 16.2

Start a new project called *UsingAtlasImages*. Compile the default code and copy the files *HandsOnAGK/Chapter16/CompositeImage.png* and *CompositeImage subimages.txt* into the project's *media* folder.

Modify *main.agc*'s code to match that given in FIG-16.9. Test and save your project.

The ImageJoiner Utility

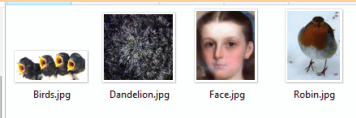
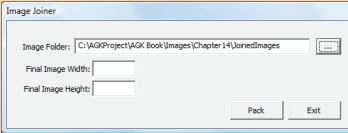
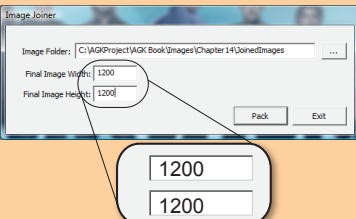

Since it can be quite time-consuming creating the atlas image and accompanying text file, the Game Creators have thoughtfully created a utility called *ImageJoiner* to do most of the work for you.

ImageJoiner is included in the AGK package. Just find the folder in which AGK has been installed (probably *C:/Program Files/The Game Creators/AGK*). In a subfolder named *Utilities* you will find the *ImageJoiner.exe*. Click on this to start up the utility.

But before you start the utility, you need to gather together all the images you want to have included in the atlas image into a single folder. Once you've done this, you are ready to run the utility. The steps involved in running the utility are shown in FIG-16.10.

FIG-16.10

Using the ImageJoiner Utility

<p>The images that are to be placed in the atlas file are first gathered into a folder which should contain no other files.</p>	<p>Next the <i>ImageJoiner</i> program can be executed and the folder containing the images is selected.</p>
	
<p>Now for the difficult bit, you have to give the dimensions of the atlas image, though it doesn't matter if you make it too large.</p>	<p>The software will then create the atlas image (in <i>png</i> format) and the corresponding text file in a subfolder called <i>result</i>.</p>
	

Notice that the composite image created contains a good deal of wasted white space. You can minimise this by taking the time to calculate the required width and height values for the atlas file from the dimensions of the images to be included.

Activity 16.3

Create a new subfolder, *UsingImageJoiner*, within your *HandsOnAGK* folder.

Copy to this folder the downloaded files *Birds.jpg*, *Dandelion.jpg*, *Face.jpg* and *Robin.jpg* from *HandsOnAGK/Chapter16*.

Examine the dimensions of the four images and determine an optimal size for the composite image.

Run *ImageJoiner* and use it to create a composite image from the four images named above. Examine the resulting composite image and *subimages.txt* file.

Rename the two files *Act16Images.png* and *Act16Images subimages.txt*.

Atlas Texture Files and Proportional Fonts

As we saw at the end of Chapter 14, it is an atlas texture file that is employed when specifying a new proportional font for use with the `Print()` statement and text objects.

It is not absolutely necessary to create an image of every possible character from ASCII 32 to ASCII 127. The image for any character that is omitted will be automatically replaced by the space character image.

In this simple example, the atlas file contains only five characters (ABWI and space). The atlas file and accompanying text file are shown in FIG-16.11.

FIG-16.11

Proportional Font Files

The actual image must use white text on a transparent background.

texture0.png

texture0 subimages.txt



```
87:0:0:98:75
66:98:0:73:75
65:171:0:73:75
32:0:75:73:75
73:0:150:25:75
```

Note that the filenames within the text file consist of the ASCII value for the character represented (i.e. 65 for a capital A) but not the *.png* file extension. So, if you have used *ImageJoiner* to create the atlas image, you will have to edit the text file to remove all of the *.png* extensions that will have automatically been included in the filenames.

FIG-16.12

Using a Proportional Font

```
rem *** Using a proportional font ***
rem *** Load the atlas image ***
LoadImage(1,"Texture0.png")
rem *** Set as the default font ***
SetTextDefaultFontImage(1)
rem *** Display some text ***
CreateText(1,"ABXWI")
SetTextSize(1,10)
Sync()
do
loop
```

Any character not given in the atlas file is replaced by the space character, so the output will produce a space between the *B* and *W* characters.

Activity 16.4

Start a new project called *ProportionalFont*. Copy the files *texture0.png* and *texture0 subimages.txt* into the *media* folder, then implement the text given in FIG-16.12.

Manipulating Images

CopyImage()

We can copy part of an existing image and place it in a new image using the `CopyImage()` statement (see FIG-16.13).

FIG-16.13

`CopyImage()`

Version 1

```
CopyImage ( id , imgId , x , y , width , height )
```

Version 2

```
integer CopyImage ( imgId , x , y , width , height )
```

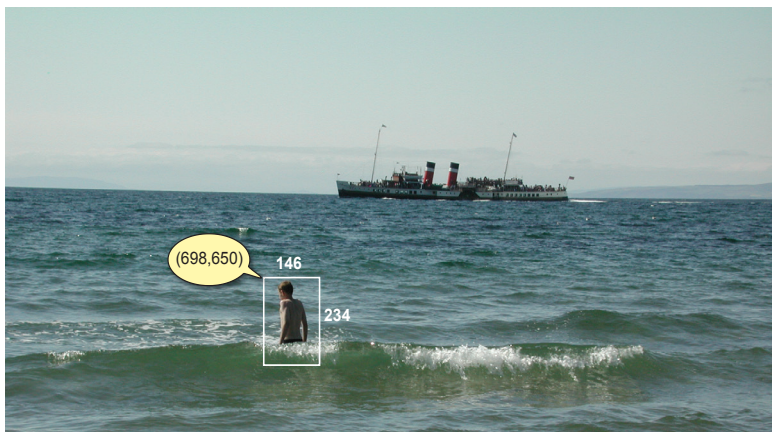
where

id	is an integer value giving the ID to be assigned to the new image.
imgId	is an integer value giving the ID of the existing image, part of which is to be copied.
x,y	are integer values giving the coordinates of the top-left corner of the area to be copied. This is given in pixels.
width	is an integer value giving the width of the area to be copied (in pixels).
height	is an integer value giving the height of the area to be copied (in pixels).

FIG-16.14

Specifying a Section of an Image

Waverley.jpg



We could copy and display the area showing the boy in the water using the program given in FIG-16.15.

FIG-16.15

Using CopyImage()

Coordinates and sizes can be obtained from paint programs such as Photoshop.

```
rem *** Copy Part of an Image ***

rem *** Load original image ***
LoadImage(1,"Waverley.jpg")

rem *** Copy a section of the image ***
CopyImage(2,1,698,650,146,234)

rem *** Assigned copied area to a sprite ***
CreateSprite(1,2)
SetSpriteSize(1,50,-1)

do
    Sync()
loop
```

Activity 16.5

Start a new project called *CopyImageSection* and implement the code given in FIG-16.15. Copy *AGKDownloads/Chapter16/Waverley.jpg* to the *media* folder.

Test and save your project.

GetImage()

If you want to copy part of screen rather than part of an image, this can be achieved using the `GetImage()` statement which will turn the captured area into another image. The statement's format is shown in FIG-16.16.

FIG-16.16

GetImage()

Version 1

`GetImage` (`id` , `x` , `y` , `width` , `height`)

Version 2

`integer` `GetImage` (`x` , `y` , `width` , `height`)

where

- | | |
|---------------|--|
| id | is an integer value giving the ID to be assigned to the new image. |
| x,y | are real values giving the coordinates of the top-left corner of the area to be copied. Use percentage or virtual pixels as appropriate. |
| width | is an integer value giving the width of the area to be copied (in percentage or virtual pixels). |
| height | is an integer value giving the height of the area to be copied (in percentage or virtual pixels). |

For example, the line

```
GetImage(2,0,0,100,50)
```


would capture the top half of the screen and store that as an image with its ID set to 2.

SaveImage()

Since an image can be created from part of another image or from capturing screen output, you may wish to save such an image to a file. This can be achieved using `SaveImage()` (see FIG-16.17).

FIG-16.17

SaveImage()

`SaveImage` (`(` `id` `,` `sfile` `)`

where

id is an integer value giving the ID of the image to be saved.

sfile is a string giving the name to be used when saving the file. The string may include path information relative to the current folder.

GetImageFilename()

The name of the file from which an image originated can be discovered using `GetImageFilename()` (see FIG-16.18).

FIG-16.18

GetImageFileName()

string `GetImageFilename` (`(` `id` `)`

where

id is an integer value giving the ID of the image.

The statement returns the name of the file in which the specified image is stored.

PrintImage()

If the device running your app has access to a printer, then you can print an image using `PrintImage()` (see FIG-16.19).

FIG-16.19

PrintImage()

`PrintImage` (`(` `id` `,` `fsize` `)`

where

id is an integer value giving the ID of the image to be printed.

fsize is a real number (0 to 100) giving the size of the image on the paper. 100 represents maximum size.

SetImageMask()

Images normally have four **channels**. Three of these channels represent the primary colours that make up the image: red, green and blue. The final channel is the transparency channel (also known as the **alpha** channel).

You can copy a channel from one image to another using the `SetImageMask()` statement (see FIG-16.20).

FIG-16.20 SetImageMask()

```
SetImageMask ( ( id , imgId , imgId , idch , isch , xoff , yoff )
```

where

- id** is an integer value giving the ID of the image to be modified.
- imgId** is an integer value giving the ID of the image from which a channel is being copied.
- idch** is an integer value (1 to 4) giving the channel to be replaced in the original image.
- isch** is an integer value (1 to 4) giving the channel to be copied.
- xoffset** is an integer value giving the horizontal offset of the new channel.
- yoffset** is an integer value giving the vertical offset of the new channel.

Perhaps the most obvious use of this statement is to copy an alpha channel from one image to another.

Starting with two images (see FIG-16.21) the program in FIG-16.22 uses the alpha channel of the second image to create a jigsaw piece.

FIG-16.21

Images Used for Masking

Image 1



Image 2



The white area of image 2 is transparent

FIG-16.22

Using SetImageMask()

```
rem *** Copy a Channel ***

rem *** Load original images ***
LoadImage(1,"Clematispiece.png")
LoadImage(2,"Jigsawpiece.png")
rem *** Create normal sprite ***
CreateSprite(1,1)
SetSpriteSize(1,25,-1)
SetSpritePosition(1,5,5)
Sync()

rem *** wait 1 second ***
Sleep(1000)

rem *** Copy the alpha channel ***
rem *** from image 2 to image 1 ***
SetImageMask(1,2,4,4,0,0)
do
    Sync()
loop
```

Activity 16.6

Start a new project called *JigsawPiece* and implement the code given in FIG-16.22. Copy files *Clematis.png* and *Jigsawpiece.png* to the *media* folder.

Test and save your project.

Image Selection from Storage

ShowChooseImageScreen()

Images already stored within a device's memory can be selected. This is done using the `ShowChooseImageScreen()` statement (see FIG-16.23).

FIG-16.23

ShowChooseImageScreen() integer `ShowChooseImageScreen()` ()

Depending on the device being used, the app may continue to run (just as it does when a sound file is being played) or it may halt while an image is selected.

The function returns 1 if the photo store was successfully opened, otherwise zero is returned.

Under Microsoft Windows, the folder *User/Pictures* is opened.

IsChoosingImage()

To discover if the user is currently in the process of selecting an image (initiated by `ShowChooseImageScreen()`) use `IsChoosingImage()` (see FIG-16.24).

FIG-16.24

IsChoosingImage() integer `IsChoosingImage()` ()

The function returns 1 while the selection process is continuing, otherwise zero is returned.

GetChosenImage()

Once the user has selected an image, we can discover the ID assigned to that image using `GetChosenImage()` (see FIG-16.25).

FIG-16.25

GetChosenImage() integer `GetChosenImage()` ()

From this point we can save the image to a file in the app's folder or assign it to a sprite.

The program in FIG-16.26 demonstrates the use of the previous three statements by allowing the user to select an image from storage and then displaying it in a sprite.

FIG-16.26

Selecting an Image

```
rem *** Select an Image from a Folder ***

rem *** If image folder found ***
if ShowChooseImageScreen() = 1
    rem *** Wait until an image has been selected ***
```

FIG-16.26

(continued)

Selecting an Image

```

rem *** Wait until an image has been selected ***
repeat
    Print("Image being selected")
    Sync()
until IsChoosingImage() = 0

rem *** Get ID assigned to selected image ***
id = GetChosenImage()

rem *** Display image in a sprite ***
CreateSprite(1,id)
SetSpriteSize(1,100,-1)
else
    rem *** Image folder not found ***
    Print("Image folder not found")
endif
Sync()
Sleep(2000)
do
    Sync()
loop

```

Activity 16.7

Start a new project called *SelectImage* and implement the code given in FIG-16.26. Test and save your project.

Using a Device's Camera

Rather than select an existing image, AGK can capture a from your device's built-in camera. The following commands are used to handle this situation.

GetCameraExists()

To check if the device running your app has a camera, use `GetCameraExists()` (see FIG-16.27).

FIG-16.27

GetCameraExists()

integer `GetCameraExists()`

The function returns 1 if a camera is available, otherwise zero is returned.

ShowImageCaptureScreen()

You can use the built-in camera to capture a photograph using the `ShowImageCaptureScreen()` statement (see FIG-16.28).

FIG-16.28

ShowImageCaptureScreen()

integer `ShowImageCaptureScreen()`

If the device does not contain a camera or the camera is not operational, this function returns 0, otherwise 1 is returned.

Depending on the device being used, the app may continue to run while the image is captured or it may halt during this time.

IsCapturingImage()

To check if the app is current performing an image capture from the camera, use `IsCapturingImage()`. This function returns 1 if `ShowImageCaptureScreen()` is currently being executed, otherwise zero is returned. The statement has the format shown in FIG-16.29).

FIG-16.29

IsCapturingImage()

integer `IsCapturingImage()` ()

GetCapturedImage()

Once the user has captured an image, the ID assigned to the image is returned by the function `GetCapturedImage()` (see FIG-16.30).

FIG-16.30

GetCapturedImage()

integer `GetCapturedImage()` ()

If the user cancels the image capture operation, this function returns zero.

The code required to capture and display an image, is very similar to the earlier program which selected an existing image. The new program is shown in FIG-16.31.

FIG-16.31

Using a Camera

```
rem *** Get an Image from the Built-In Camera ***

rem *** If camera exists ***
if GetCameraExists() = 1
    rem *** If camera software operational ***
    if ShowImageCaptureScreen() = 1
        rem *** Wait until an image has been captured ***
        repeat
            until IsCapturingImage() = 0

        rem *** Get ID assigned to selected image ***
        id = GetCapturedImage()

        rem *** Display image in a sprite ***
        CreateSprite(1,id)
        SetSpriteSize(1,100,-1)
    else
        rem *** Camera not operational ***
        Print("Image folder not found")
    endif
else
    rem *** No camera exists ***
    Print("No camera detected")
endif
Sync()
Sleep(2000)
do
    Sync()
loop
```

Activity 16.8

Start a new project called *CaptureImage* and implement the code given in FIG-16.31. Test your program. Modify the program to save the captured image to a file called *saved.jpg*. Test and save your project.

Mapping Images to Sprites

Although it is possible to create a sprite without an associated image, under most circumstances we will want to use a sprite to display an image. Exactly how that image is mapped onto the sprite can be controlled by various commands.

SetDefaultMagFilter()

When the image to be mapped to a sprite is larger than the sprite on which it will be displayed (for example, if the image was 1024 x 1024 pixels and the sprite occupied 128x 128 pixels on the screen), then we can control how the larger image is “squeezed” onto the sprite using the `SetDefaultMagFilter()` (see FIG-16.32).

FIG-16.32

SetDefaultMagFilter()

`SetDefaultMagFilter ((iop))`

where

iop is an integer value (0 or 1) specifying the mapping option to be used. Option 0 uses a “nearest pixel” mapping which tends to give a sharper but blocky image; option 1 uses average pixel mapping which gives a slightly more blurred image.

Using this function affects all further mapping when the image is larger than the sprite. By default, your program will use option 1 mapping.

SetDefaultMinFilter()

When the image to be mapped to a sprite is smaller than the sprite, then we can control how the smaller image is “stretched” onto the sprite using the `SetDefaultMinFilter()` (see FIG-16.33).

FIG-16.33

SetDefaultMinFilter()

`SetDefaultMinFilter ((iop))`

where

iop is an integer value (0 or 1) specifying the mapping option to be used. Option 0 uses a “nearest pixel” mapping; option 1 uses average pixel mapping.

Using this function affects all further mapping when the image is smaller than the sprite. By default, your program will use option 1 mapping.

SetImageMagFilter()

Although the `SetDefaultMagFilter()` statement sets the mapping used for all larger-than-sprite images, you can override that mapping for individual images using the `SetImageMagFilter()` statement (see FIG-16.34).

FIG-17.34

SetImageMinFilter()

`SetImageMagFilter ((id , iop))`

where

id is an integer value giving the ID of the image whose mapping is to be set.

iop is an integer value (0 or 1) specifying the mapping option to be used. Option 0 uses a “nearest pixel” mapping; option 1 uses average pixel mapping.

SetImageMinFilter()

When the image to be mapped to a sprite is smaller than the sprite, then we can control how the smaller image is “stretched” onto the sprite using the `SetDefaultMinFilter()` (see FIG-16.35).

FIG-16.35

SetImageMinFilter()

`SetImageMinFilter ((id , iop)`

where

id is an integer value giving the ID of the image whose mapping is to be set.

iop is an integer value (0 or 1) specifying the mapping option to be used. Option 0 uses a “nearest pixel” mapping; option 1 uses average pixel mapping.

The program in FIG-16.36 makes use of the four filter statements and shows how they affect the final image displayed.

FIG-16.36

Using Image Filters

```
rem *** Using Image Filters ***

rem *** Set average pixel mapping ***
SetDefaultMagFilter(1) //This is the default
SetDefaultMinFilter(1) //This is the default

rem *** Load small (128x128) image ***
LoadImage(1,"XSmall.png")
LoadImage(2,"XSmall.png")
rem *** Load large (1024x1024) image ***
LoadImage(3,"XLarge.png")
LoadImage(4,"XLarge.png")

rem *** Override mapping for images 2,4 ***
SetImageMagFilter(2,0)
SetImageMinFilter(4,0)

rem *** Create large sprites for small images ***
CreateSprite(1,1)
SetSpriteSize(1,50,-1)
CreateSprite(2,2)
SetSpriteSize(2,50,-1)
SetSpritePosition(2,50,0)
rem *** Create small sprites for large images ***
CreateSprite(3,3)
SetSpriteSize(3,20,-1)
SetSpritePosition(3,30,50)
CreateSprite(4,4)
SetSpriteSize(4,20,-1)
SetSpritePosition(4,50,50)

do
    Sync()
loop
```

Activity 16.9

Start a new project called *ImageFilters* and implement the code given in FIG-16.36. Copy the file *XLarge.png* and *XSmall.png* to the project's *media* folder.

Test your program and observe the effects of each mapping. Save your project.

SetGenerateMipmaps()

Start with an image of a given size and then make a copy of that image but at half the width and height. Now do the same thing to the second image, creating a third even smaller image and then continue the process until several images have been created. This is the idea behind a **mipmap** an example of which is shown in FIG-16.37.

FIG-16.37

An Example of a Mipmap



Mipmaps make the job of mapping an image onto a sprite just a little easier since AGK will choose the image within the mipmap which is closest in size to the sprite and this reduces the processing requirements. They may be particularly useful when a sprite changes size during the execution of a program.

AGK will automatically create a mipmap for every image that is loaded if the `SetGenerateMipmaps()` statement is executed before any image is loaded. The format of this statement is shown in FIG-16.38.

FIG-16.38

`SetGenerateMipmaps()`

`SetGenerateMipmaps ((iop))`

where

iop is an integer value (0 or 1) which determines if mipmaps are to be created (0: no mipmaps, 1: create mipmaps).

Using a mipmap increases the storage requirements for any image by about 33%.

AGK's default option is not to create mipmaps.

Summary

- Use `GetImageExists()` to check that an image of a specified ID currently exists.
- Use `GetImageWidth()` to discover the width (in pixels) of an image.
- Use `GetImageHeight()` to discover the height (in pixels) of an image.

- An atlas texture image is a composite of two or more images.
- An atlas texture image must be accompanied by a text file giving details of the composite images within the atlas image.
- The images within an atlas image can be extracted as standard image components using the `LoadSubImage()` statement.
- The ImageJoiner utility (supplied with AGK) can be used to help construct an atlas image and the accompanying text file.
- If you use ImageJoiner to create an atlas file containing a new font, the accompanying text file must be edited so that the `.png` extension is removed from every filename.
- Use `CopyImage()` to create a new image from a section of an already loaded image.
- Use `GetImage()` to create a new image from a section of the screen display.
- Use `SaveImage()` to save an image to a file.
- Use `GetImageFilename()` to discover the name of the file from which an image has been loaded.
- Use `PrintImage()` to print an image.
- Use `SetImageMask()` to copy a channel from one image to another.
- Use `ShowChooseImageScreen()` to display the images available on a device.
- Use `IsChoosingImage()` to determine if the user is currently in the process of selecting a stored image.
- Use `GetChosenImage()` to load the selected image into an AGK image resource and assign it an ID.
- Use `GetCameraExists()` to check if the device running your app has a camera.
- Use `ShowImageCaptureScreen()` to activate the camera app on your device.
- Use `IsCapturingImage()` to determine if the user is currently in the process of capturing an image.
- Use `GetCapturedImage()` to assign the image taken by the camera to an AGK image resource and assign it an ID.
- Use `SetDefaultMagFilter()` to specify how large images should be mapped to smaller sprites.
- Use `SetDefaultMinFilter()` to specify how small images should be mapped to larger sprites.
- Use `SetImageMagFilter()` to modify the mapping of a specific larger image to a smaller sprite.
- Use `SetImageMagFilter()` to modify the mapping of a specific smaller image to a larger sprite.
- Use `SetGenerateMipmaps()` to create mipmaps for all images used in a program.

Solutions

Activity 16.1

Code for *ImageProperties*:

```
rem *** Image Properties ***

rem *** Load image ***
LoadImage(1,"Size.png")
rem *** 50% chance of deleting the image ***
if Random(1,2) = 1
    DeleteImage(1)
endif
rem *** IF image loaded ***
if GetImageExists(1)=1
    rem *** Display details ***
    Print("Image found")
    Print("Image width  = "+Str(GetImageWidth(1)))
    Print("Image height = "+Str(GetImageHeight(1)))
else
    rem *** ELSE display message ***
    Print("Image not found")
endif
Sync()
do
loop
```

```
endif
else
    rem *** No camera exists ***
    Print("No camera detected")
endif
Sync()
Sleep(2000)
do
    Sync()
loop
```

Activity 16.9

No solution required.

Activity 16.2

No solution required.

Activity 16.3

No solution required.

Activity 16.4

No solution required.

Activity 16.5

No solution required.

Activity 16.6

No solution required.

Activity 16.7

No solution required.

Activity 16.8

Modified code for :

```
rem *** Get an Image from the Built-In Camera ***

rem *** If camera exists ***
if GetCameraExists() = 1
    rem *** If camera software operational ***
    if ShowImageCaptureScreen() = 1
        rem *** Wait until an image has been
        captured ***
        repeat
        until IsCapturingImage() = 0

        rem *** Get ID assigned to selected image
        ***
        id = GetCapturedImage()

        rem *** Display image in a sprite ***
        CreateSprite(1,id)
        SetSpriteSize(1,100,-1)

        rem *** Save Image to a file ***
        SaveImage(id,"saved.jpg")
    else
        rem *** Camera not operational ***
        Print("Image folder not found")
    end
end
```

17

Sprites

In this Chapter:

- ☐ Adjusting a Sprite's Attributes
- ☐ Modifying How an Image Maps to a Sprite
- ☐ Repositioning a Sprite's Origin
- ☐ Grouping Sprites
- ☐ Changing a Sprite's Bounding Area
- ☐ Checking for Sprite Collision
- ☐ Sprite Groups and Categories
- ☐ Handling Moving Sprites at the Edge of the Screen
- ☐ Setting a Program's Frame Rate
- ☐ Ray Casting
- ☐ A Jigsaw Puzzle Game

Introduction

There are many more sprite commands available to us than the few already covered in Chapter 6. These additional commands allow us to perform various operations such as rotating a sprite, remapping the image on the sprite's surface, detecting sprite collisions and even calculating the distance between two sprites.

Review

The following sprite-related statements were covered in Chapter 6:

	<code>CreateSprite(id,imgId)</code>	This statement creates a sprite using a previously loaded image. <i>id</i> is an integer giving the ID to be assigned to the sprite; <i>imgId</i> is the ID previously assigned to the image.
The term <i>int</i> is used here as a shortened form of the word <i>integer</i> .	<code>int CreateSprite(imgId)</code>	This statement creates a sprite using image <i>imgId</i> and returns the ID assigned to the sprite.
	<code>SetSpriteSize(id,w,h)</code>	This statement sets the width and height of sprite, <i>id</i> . <i>w</i> and <i>h</i> are real values giving the width and height. These are given as percentages or virtual pixels depending on the screen setup. Either <i>w</i> or <i>h</i> can be set to -1 to ensure that the original aspect ratio of the image assigned to the sprite is maintained.
	<code>SetSpritePosition(id,x,y)</code>	This statement positions the sprite, <i>id</i> , on the screen. The top left corner of the sprite is positioned at coordinates (<i>x</i> , <i>y</i>). <i>x</i> and <i>y</i> are real values given as either a percentage or as virtual coordinates. When first created, a sprite's top-left corner is at position (0,0) - the top left corner of the screen
	<code>SetSpriteDepth(id,idepth)</code>	This statement sets the depth of the sprite, <i>id</i> . The depth setting determines how overlapping sprites appear. A sprite with a lower depth value will appear "in front of" a sprite with a larger depth value. The default depth setting for all sprites is 10. The order of sprites with the same depth setting is determined by the order in which they were created - newly created sprites appear "above" older sprites. The depth setting can range from 0 (front) to 10,000 (back).
	<code>CloneSprite(id,idcopied)</code>	This statement creates an exact copy of an existing sprite. <i>id</i> is the ID assigned to the new sprite; <i>idcopied</i> is the ID of the sprite to be cloned.
	<code>SetSpriteVisible(id,iv)</code>	This statement sets the visibility of sprite <i>id</i> . If <i>iv</i> is 1 the sprite is visible; if <i>iv</i> is 0, the sprite is hidden.

`DeleteSprite(id)`

This statement deletes the sprite, *id*. Deleting any item when it is no longer required frees up resources and may increase a program's speed.

Activity 17.1

Start a new project called *ControllingSprites*. Compile the default code and copy the file *Arrow.png* from *AGKDownloads/Chapter17* to the *media* folder. Place the following code in *main.agc*.

```
rem *** Controlling a sprite ***
rem *** Load sprite image ***
LoadImage(1,"Arrow.png",0)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Size sprite ***
SetSpriteSize(1,7,-1)
rem *** Position sprite ***
SetSpritePosition(1,50,50)
Sync()
do
loop
```

Test and save your project.

Other Sprite Statements

Most of the remaining sprite statements cover new features, but a few relate back to attributes covered in Chapter 6.

GetSpriteExists()

We can determine if a sprite with a given ID currently exists using the `GetSpriteExists()` statement (see FIG-17.1).

FIG-17.1

`GetSpriteExists()`

integer `GetSpriteExists` (`id`)

where:

id is the integer value giving the ID to be checked.

The function returns 1 if the sprite exists, otherwise zero is returned.

GetSpriteVisible()

To find out if an existing sprite is visible, use `GetSpriteVisible()` (see FIG-17.2).

FIG-17.2

`GetSpriteVisible()`

integer `GetSpriteVisible` (`id`)

where:

id is the integer value giving the ID of the sprite to be checked.

The function returns 1 if the sprite is visible, otherwise zero is returned.

GetSpriteDepth()

`GetSpriteDepth()` returns the depth setting of a sprite. The statement's format is shown in FIG-17.3.

FIG-17.3

GetSpriteDepth()

integer `GetSpriteDepth` ((`id`))

where:

id is the integer value giving the ID of the sprite to be checked.

The function returns the layer on which the specified sprite has been placed. By default, all sprites are placed on layer 10. Sprites on a higher layer (layer 0 is the highest layer, layer 10,000 is the lowest layer) will appear to be positioned on top of those on a lower layer.

SetSpriteScale()

The best way to set your sprite to the size required is to use the `SetSpriteSize()` statement (see Chapter 6), but you can also resize a sprite using the `SetSpriteScale()` statement (see FIG-17.4).

FIG-17.4

SetSpriteScale()

`SetSpriteScale` ((`id` , `xscale` , `yscale`))

where:

id is the integer value giving the ID of the sprite.

xscale is a real number giving the multiplication factor to be applied along the width of the sprite.

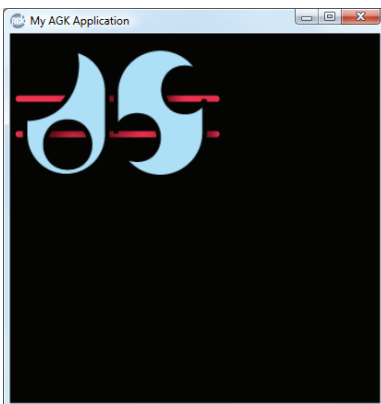
yscale is a real number giving the multiplication factor to be applied along the height of the sprite.

Once the size of a sprite has been set using `SetSpriteSize()`, this is assumed to be the sprite's "normal" size. Using `SetSpriteScale()` with a scale factor of 1.0 for both `xscale` and `yscale` will leave the image size unchanged. Values smaller than 1 will shrink the sprite; those larger than 1 will increase its size. The position of the sprite's top left corner remains fixed during scaling. FIG-17.5 shows a sprite with its initial size set to 60% of the screen width. The sprite is then reduced to 50% of its size in both the *x* and *y* directions.

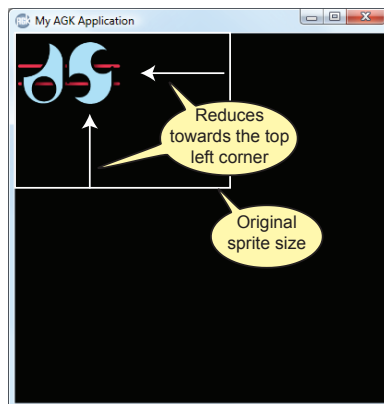
FIG-17.5

Result of Using
`SetSpriteScale()`

Sprite after `SetSpriteSize(1,60,-1)`



Sprite after `SetSpriteScale(1, 0.5, 0.5)`



SetSpriteAngle()

A sprite can be rotated by using the `SetSpriteAngle()` statement which has the format shown in FIG-17.6.

FIG-17.6

SetSpriteAngle()

`SetSpriteAngle` ((`id` , `fangle`))

where:

- id** is the integer value previously assigned as the ID of the sprite.
- fangle** is a real number giving the angle to which the sprite is to be rotated. The angle is given in degrees. The sprite will be rotated about its centre.

Activity 17.2

Modify your *ControllingSprites* project by deleting the last three lines and replacing them with the code:

```
degrees# = 0
do
    degrees# = degrees# +1
    SetSpriteAngle(1,degrees#)
    Sync()
loop
```

Check that the sprite rotates and then save your project.

SetSpriteAngleRad()

When rotating a sprite, the angle may be given in radians, rather than degrees, by using the `SetSpriteAngleRad()` statement (see FIG-17.7)

FIG-17.7

SetSpriteAngleRad()

`SetSpriteAngleRad` ((`id` , `fangle`))

where:

- id** is the integer value previously assigned as the ID of the sprite.
- fangle** is a real number giving the angle to which the sprite is to be rotated. The angle is given in radians. The sprite will be rotated about its centre.

GetSpriteAngle() and GetSpriteAngleRad()

To discover a sprite's current angle of rotation, you can use either `GetSpriteAngle()` - which returns the angle in degrees - or `GetSpriteAngleRad()` which returns the angle in radians. FIG-17.8 shows the format of both statements.

FIG-17.8

GetSpriteAngle()

float `GetSpriteAngle` ((`id`))

GetSpriteAngleRad()

float `GetSpriteAngleRad` ((`id`))

where:

- id** is the integer value previously assigned as the ID of the sprite.

SetSpriteColor()

You can set the colour of a sprite - overriding the colour of the image used - by executing the `SetSpriteColor()` statement. This statement can also change the transparency of the sprite. The `SetSpriteColor()` statement's format is shown in FIG-17.9.

FIG-17.9

SetSpriteColor()

`SetSpriteColor (id , ired , igreen , iblue , itrans)`

where:

- id** is the integer value previously assigned as the ID of the sprite.
- ired** is an integer value giving the intensity of the red component to be used when changing the sprite's colour. This value should lie in the range zero (no red) to 255 (full red).
- igreen** is an integer value giving the intensity of the green component to be used. This value should lie in the range zero to 255.
- iblue** is an integer value giving the intensity of the blue component to be used. This value should lie in the range zero to 255.
- itrans** is an integer value giving the transparency setting for the sprite. This value should lie in the range zero (invisible) to 255 (opaque).

SetSpriteColorRed(), SetSpriteColorGreen(), SetSpriteColorBlue() and SetSpriteColorAlpha()

Rather than set all the colour attributes of a sprite using the `SetSpriteColor()` statement, individual attributes can be changed using one of four commands: `SetSpriteColorRed()`, `SetSpriteColorGreen()`, `SetSpriteColorBlue()` and `SetSpriteColorAlpha()`.

The format for each of these four commands is shown in FIG-17.10.

FIG-17.10

SetSpriteColorRed()
SetSpriteColorGreen()
SetSpriteColorBlue()
SetSpriteColorAlpha()

`SetSpriteColorRed (id , ired)`

`SetSpriteColorGreen (id , igreen)`

`SetSpriteColorBlue (id , iblue)`

`SetSpriteColorAlpha (id , itrans)`

where:

- id** is the integer value previously assigned as the ID of the sprite.
- ired** is an integer value giving the intensity of the red component to be used when changing the sprite's colour. This value should lie in the range zero (no red) to 255 (full red).
- igreen** is an integer value giving the intensity of the green component to be used. This value should lie in the range zero to 255.

ibblue is an integer value giving the intensity of the blue component to be used. This value should lie in the range zero to 255.

itrans is an integer value giving the transparency setting for the sprite. This value should lie in the range zero (invisible) to 255 (opaque).

Activity 17.3

Modify *ControllingSprites*, changing the red component of the sprite to 0.

Test and save your project.

GetSpriteColorRed(), GetSpriteColorGreen(), GetSpriteColorBlue() and GetSpriteColorAlpha()

To discover the current colour settings for a sprite, you can use the commands `GetSpriteColorRed()`, `GetSpriteColorGreen()`, `GetSpriteColorBlue()`, and `GetSpriteColorAlpha()`.

The format for each of these commands is shown in FIG-17.11.

FIG-17.11

`GetSpriteColorRed()`
`GetSpriteColorGreen()`
`GetSpriteColorBlue()`
`GetSpriteColorAlpha()`

integer `GetSpriteColorRed` ((id))
integer `GetSpriteColorGreen` ((id))
integer `GetSpriteColorBlue` ((id))
integer `GetSpriteColorAlpha` ((id))

where:

id is the integer value previously assigned as the ID of the sprite.

Each function returns a value in the range 0 to 255.

GetSpriteHitTest()

To find out if a specified point on the screen is within the area of a specific sprite, you can use the `GetSpriteHitTest()` statement. The statement has the format shown in FIG-17.12.

FIG-17.12

`GetSpriteHitTest()`

integer `GetSpriteHitTest` ((id , x , y))

where:

id is the integer value previously assigned as the ID of the sprite.

x,y are a pair of real numbers giving the coordinates of the point to be tested. This should be in percentage or virtual coordinates depending on the system being used.

If point (x,y) lies within the area of the sprite *id*, then the function returns 1, otherwise zero is returned.

GetSpriteHit()

When several sprites appear on screen at the same time, finding out which sprite has been hit using `GetSpriteHitTest()` will involve using a `for` loop to cycle through each sprite ID (this assumes the ID values are contiguous). For example, a typical code snippet would be:

```
for c = lowestID to highestID
  if GetSpriteHitTest(c, GetPointerX(), GetPointerY()) = 1
    rem *** do something ***
  endif
next c
```

This is a long and awkward approach.

As an alternative, we can make use of the `GetSpriteHit()` statement which returns the ID of the sprite at a specified point. The `GetSpriteHit()` statement has the format shown in FIG-17.13.

FIG-17.13

integer `GetSpriteHit` ((`x` , `y`)

`GetSpriteHit()`

where:

`x,y` are a pair of real values giving the coordinates of the screen position to be checked.

If the specified point (`x,y`) is over a sprite, then the ID of that sprite is returned. If no sprite is under the point, then zero is returned. If more than one sprite is at the specified point, then the “top-most” sprite’s ID is returned.

Used in conjunction with `GetPointerX()` and `GetPointerY()`, we can use the `GetSpriteHit()` statement to detect user clicks/presses on a sprite using code of the form:

```
if GetSpriteHit(GetPointerX(),GetPointerY()) <> 0
  rem *** Do something ***
endif
```

The program in FIG-17.14 uses three sprites (these could represent buttons in a game) which change colour when pressed and revert to their original colour when released. The program makes use of the following logic:

```
Load images
Create, resize and position sprites
Get the current red setting for the first sprite
DO
  IF pointer pressed THEN
    IF pointer over a sprite THEN
      Change sprite's red tint
      Set oldID to sprite's ID
      Update screen
    ENDIF
  ELSE
    IF oldID isn't zero THEN
      Reset sprite's red tint
      Set oldID to zero
      Update screen
    ENDIF
  ENDIF
LOOP
```

FIG-17.14

Using GetSpriteHit()

```

rem *** Button Sprites ***
rem *** Grey background ***
SetClearColor(200,200,200)
rem *** Load sprite images ***
LoadImage(1,"Right.png",0)
LoadImage(2,"Left.png",0)
LoadImage(3,"Fire.png",0)
rem *** Create sprites ***
CreateSprite(1,1)
CreateSprite(2,2)
CreateSprite(3,3)
rem *** Size sprite ***
SetSpriteSize(1,12,-1)
SetSpriteSize(2,12,-1)
SetSpriteSize(3,12,-1)
rem *** Position sprite ***
SetSpritePosition(1,87,92)
SetSpritePosition(2,72,92)
SetSpritePosition(3,1,92)
rem *** Get red tint for first sprite ***
red = GetSpriteColorRed(1)
do
    rem *** IF pointer down ***
    if GetPointerState() = 1
        rem *** IF press if over a sprite ***
        id = GetSpriteHit(GetPointerX(),GetPointerY())
        if id <> 0
            rem *** Set sprite's tint ***
            SetSpriteColorRed(id,200)
            rem *** Remember sprite's ID ***
            oldid = id
            rem *** Update screen ***
            Sync()
        endif
    else
        rem *** IF a sprite is tinted ***
        if oldid <> 0
            rem *** Reset its tint ***
            SetSpriteColorRed(oldid,red)
            rem *** No sprite currently tinted ***
            oldid = 0
            rem *** Update screen ***
            Sync()
        endif
    endif
    Sync()
loop

```

Activity 17.4

Create a new project called *ShootingGame*. Compile the default code and copy the files *Right.png*, *Left.png* and *Fire.png* from *AGKDownloads/Chapter17* to the projects's *media* folder. Enter and test the code given in FIG-17.14. Save your project.

SetSpriteX()

Although we can use `SetSpritePosition()` to place a sprite at any point on the screen, when we wish to move a sprite horizontally we can make use of the

`SetSpriteX()` statement (see FIG-17.15).

FIG-17.15

`SetSpriteX()`

`SetSpriteX (id , x)`

where:

- id** is the integer value previously assigned as the ID of the sprite.
- x** is a real number giving the new x-coordinate of the sprite's top left corner. Coordinates are given as a percentage or in virtual coordinates depending on the screen setup.

SetSpriteY()

To set only the y-coordinate of a sprite (allowing vertical movement) use the `SetSpriteY()` statement (see FIG-17.16).

FIG-17.16

`SetSpriteY()`

`SetSpriteY (id , y)`

where:

- id** is the integer value previously assigned as the ID of the sprite.
- y** is a real number giving the new y-coordinate of the sprite's top left corner. Coordinates are given as a percentage or in virtual coordinates depending on the screen setup.

GetSpriteX() and GetSpriteY()

The current position of a sprite can be obtained using the `GetSpriteX()` and `GetSpriteY()` statements which return the *x* and *y* coordinates respectively. The format for each of these statements is shown in FIG-17.17.

FIG-17.17

`GetSpriteX()`
`GetSpriteY()`

float `GetSpriteX (id)`

float `GetSpriteY (id)`

where:

- id** is the integer value previously assigned as the ID of the sprite.

Activity 17.5

Make the following modifications to your *ShootingGame* project:

Add a sprite containing *Arrow.png* with a width setting of 6. The arrow should be pointing upwards and the sprite positioned at 46,80.

When the right button is pressed, the arrow should move 1 place to the right; when the left button is pressed, the arrow should move one place to the left.

Test your code. What happens when the arrow reaches the edge of the screen? Modify the code so that the arrow cannot move off the edge of the app window.

GetSpriteWidth() and GetSpriteHeight()

We can also determine the width and height of a sprite using the `GetSpriteWidth()` and `GetSpriteHeight()` statements (see FIG-17.18). The value returned by each of these functions is given as a percentage or in virtual pixels as appropriate.

FIG-17.18

`GetSpriteWidth()`

`GetSpriteHeight()`

float `GetSpriteWidth` ((id))

float `GetSpriteHeight` ((id))

where:

id is the integer value previously assigned as the ID of the sprite.

GetSpriteImageID()

The ID of the image that has been assigned to a sprite can be discovered using the `GetSpriteImageID()` statement (see FIG-17.19).

FIG-17.19

`GetSpriteImageID()`

integer `GetSpriteImageID` ((id))

where:

id is the integer value previously assigned as the ID of the sprite.

SetSpriteImage()

The image used by a sprite is set up when the sprite is first created, but you can change the image at a later stage in the program using the `SetSpriteImage()` statement (see FIG-17.20).

FIG-17.20

`SetSpriteImage()`

`SetSpriteImage` ((id , imageId))

where:

id is the integer value previously assigned as the ID of the sprite.

imgId is an integer value giving the ID of the image to be assigned to the sprite.

The program in FIG-17.21 displays a sprite then changes its image if the pointer is pressed while over the sprite.

FIG-17.21

Using `SetSpriteImage()`

```
rem *** Swap sprite image ***

rem *** Load Images ***
LoadImage(1,"Round.png")
LoadImage(2,"Square.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
do
    rem *** If pointer pressed and over sprite ***
```



FIG-17.21

(continued)

Using `SetSpriteImage()`

```

    if GetSpriteHit(GetPointerX(),GetPointerY()) and
       ↳GetPointerPressed()
       rem *** Change to other image ***
       SetSpriteImage(1,2)
    endif
    Sync()
loop

```

Activity 17.6

Start a new project called *SwapImage*. Copy the files *Round.png* and *Square.png* from *AGKDownloads/Chapter17* and paste them to the new project's *media* folder.

Implement and test the code given in FIG-17.21.

Modify the code so that the sprite switches between the two images each time it is pressed.

When the new image to be placed within the sprite has a different width to height ratio from the first image, then the new image will be distorted in order to fit the sprite's original space; the sprite will not be resized to accommodate the new image.

Activity 17.7

Copy the file *FourCircles.png* from *AGKDownloads/Chapter17* to your *SwapImage* project's *media* folder.

Open the image to see its contents.

Modify *SwapImage* so that the *FourCircles* image is used in place of *Square.png*.

What happens when the four circle image is displayed?

To overcome the distortion problem, all that is required is a subsequent call to `SetSpriteSize()`.

Activity 17.8

Copy the `SetSpriteSize()` used near the start of *SwapImage* and duplicate it immediately after the `SetSpriteImage()` statement.

What happens this time when the four circle image is displayed?

Normally, the best way to handle a sprite which displays different images at different times, is to create an animated sprite (see next chapter) but using `SetSpriteImage()` may be useful when those images are of different sizes.

SetSpriteTransparency()

Normally, transparency is set up when an image is being loaded, with the option to use a transparent background in the original image or make black pixels transparent.

However, the parts of an image which are normally transparent can be made opaque using the `SetSpriteTransparency()` statement (see FIG-17.22).

FIG-17.22

`SetSpriteTransparency()`

`SetSpriteTransparency (id , itrans)`

where:

id is the integer value previously assigned as the ID of the sprite.

itrans is an integer value (0 or 1) setting the transparency mode of the image. 0: no transparency; 1: transparency (as determined by the original sprite image).

For a truly rectangular image, transparency is not required and having transparency switched off will give a boost to performance.

The program in FIG-17.23 shows a sprite with transparency off and then, after two seconds, it is switched on.

FIG-17.23

Using
`SetSpriteTransparency()`

```
rem *** Sprite transparency ***
rem *** Set grey background ***
SetClearColor(120,120,120)
Sync()
rem *** Load Image ***
LoadImage(1,"Round.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
SetSpritePosition(1,50,50)
rem *** Sprite transparency off ***
SetSpriteTransparency(1,0)
Sync()
Sleep(2000)
rem *** Sprite transparency on ***
SetSpriteTransparency(1,1)
Sync()
Sleep(2000)
do
loop
```

Activity 17.9

Start a new project called *SpriteTransparency*. Copy the file *Round.png* (which you used in the last project) to the project's *media* folder.

Implement and test the code given in FIG-17.23.

SetSpriteFlip()

We can reflect the image shown in a sprite vertically, horizontally, or both using the `SetSpriteFlip()` statement (see FIG-17.24).

FIG-17.24

`SetSpriteFlip()`

`SetSpriteFlip (id , ihorz , iver)`

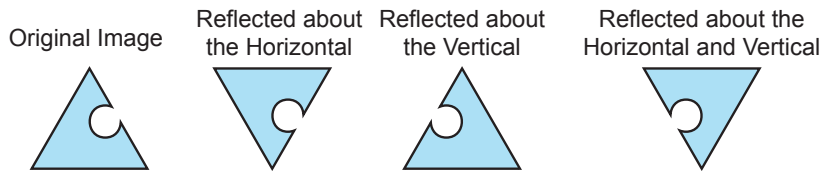
where:

id	is the integer value previously assigned as the ID of the sprite.
ihorz	is an integer value (0 or 1) which determines if the image is to be reflected about the horizontal (0: do not reflect, 1: reflect).
ivert	is an integer value (0 or 1) which determines if the image is to be reflected about the vertical (0: do not reflect, 1: reflect).

The results of reflecting an image in each direction are shown in FIG-17.25.

FIG-17.25

Flip Options



The program in FIG-17.26 shows a sprite using each reflection with accompanying descriptive text.

FIG-17.26

Using SetSpriteFlip()

```
rem *** Flipping Sprites ***
rem *** Load Image ***
LoadImage(1,"Triangle.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
SetSpritePosition(1,44,50)
rem *** Create descriptive text ***
CreateText(1,"Original Image")
SetTextAlignment(1,1)
SetTextPosition(1,50,40)
Sync()
Sleep(2000)
rem *** Reflect about the horizontal ***
SetTextString(1,"Reflected about Horizontal")
SetSpriteFlip(1,1,0)
Sync()
Sleep(2000)
rem *** Reset sprite ***
SetTextString(1,"Original Image")
SetSpriteFlip(1,0,0)
Sync()
Sleep(2000)
rem *** Reflect about the vertical ***
SetTextString(1,"Reflected about Vertical")
SetSpriteFlip(1,0,1)
Sync()
Sleep(2000)
rem *** Reset sprite ***
SetTextString(1,"Original Image")
SetSpriteFlip(1,0,0)
Sync()
Sleep(2000)
rem *** Reflect about the horizontal and vertical ***
SetTextString(1,"Reflected about"+chr(10)+"Horizontal and
Vertical")
SetSpriteFlip(1,1,1)
Sync()
do
loop
```


Activity 17.10

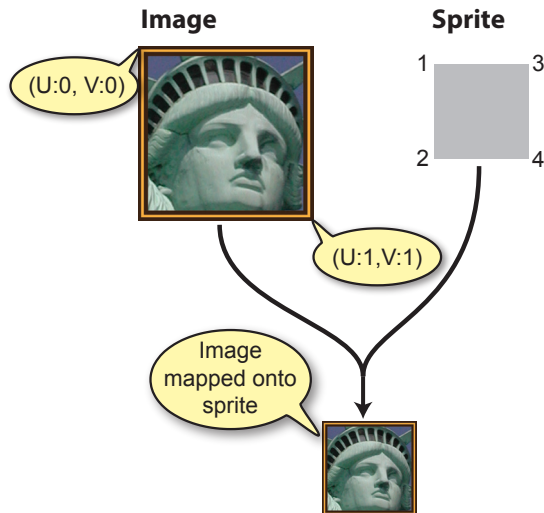
Create a new project called *FlippingSprites* and copy *AGKDownloads/Chapter17/Triangle.png* to the project's *media* file. Implement, test and save the code given in FIG-17.26 above.

SetSpriteUVScale()

When an image is placed within a sprite, we might think of this as something akin to pasting a picture (the image) onto a piece of card (the sprite). Normally, when we fix the image to the card, the top left of the image is placed at the top-left of the sprite and the bottom-right of the image at the bottom right of the sprite. When we describe exactly how the image is “mapped” onto the sprite, we use the coordinates (0,0) to represent the top left of the image and (1,1) to represent the bottom right. These values are used irrespective of the image's actual size or width to height ratio. Since we already use the letters X and Y to represent positions in normal two-dimensional space, we use U and V to represent width and height within the image. FIG-17.27 demonstrates this idea.

FIG-17.27

How an Image is Mapped to a Sprite



As you can see from FIG-17.27, (0,0) on the image is mapped to corner 1 of the sprite; (0,1) to corner 2; (1,0) to corner 3; and (1,1) to corner 4.

AGK contains various commands that affect how this mapping of the image to the sprite is performed.

Using the `SetSpriteUVScale()` we can modify how the image is stretched over the sprite. With the correct parameter values, we can shrink the image so it occupies only part of the surface of the sprite or enlarge it so that only part of the image is used on the sprite. `SetSpriteUVScale()` has the format shown in FIG-17.28.

FIG-17.28

SetSpriteUVScale()

`SetSpriteUVScale` ((`id` , `Uscale` , `Vscale`)

where:

id is the integer value previously assigned as the ID of the sprite.

Uscale is a real value giving the width of the image on the sprite.

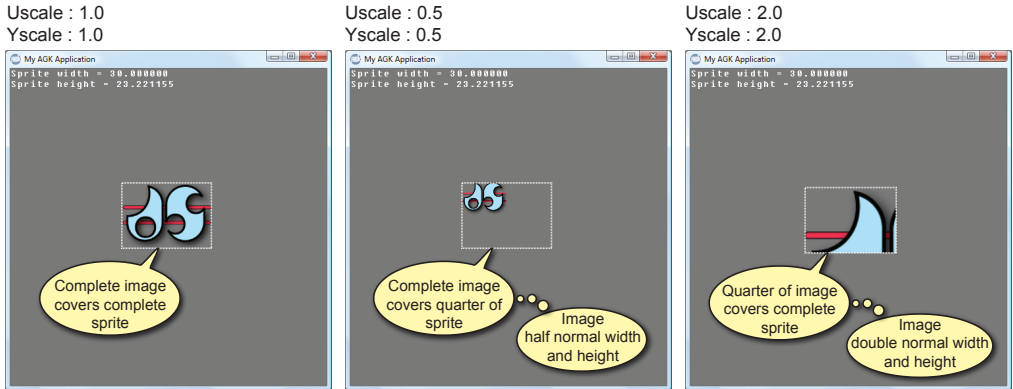
0: no width; 0.5: image stretches over half the width of the sprite); 1: normal (image over full width of sprite); 2: image is twice the width of the sprite (only half the image is seen).

Vscale is a real value which determines the height of the image on the sprite.

FIG-17.29

UV Scaling Effects

FIG-17.29 shows the result of various scale options on the appearance of a sprite. The dotted outline indicates the size and position of the sprite.



The program in FIG-17.30 demonstrates the use of UV scaling.

FIG-17.30

Using SetSpriteUVScale()

```
rem *** UV Scaling ***
rem *** Grey screen ***
ClearColor(120,120,120)
Sync()
rem *** Load image ***
LoadImage(1,"DS.png")
rem *** Create, size and position sprite ***
CreateSprite(1,1)
SetSpriteSize(1,30,-1)
SetSpritePosition(1,35,35)
rem *** Normal UV scaling ***
SetSpriteUVScale(1,1,1)
Sync()
Sleep(2000)
rem *** Half scaling ***
SetSpriteUVScale(1,0.5,0.5)
Sync()
Sleep(2000)
rem *** Double scaling ***
SetSpriteUVScale(1,2,2)
Sync()
do
loop
```

Activity 17.11

Start a new project named *UVScaling* and implement the code given in FIG-17.30. Remember to copy *AGKDownloads/Chapter17/DS.png* to the *media* folder.

Test and save your project.

SetSpriteUOffset()

Another option available when mapping an image to a sprite is to offset the image so that the top-left of the image does not appear at the top-left of the sprite. This is achieved using the `SetSpriteUOffset()` statement (see FIG-17.31).

FIG-17.31

`SetSpriteUOffset()`

`SetSpriteUOffset ((id , Uoffset , Voffset)`

where:

id is the integer value previously assigned as the ID of the sprite.

Uoffset, Voffset

are a set of real values giving the coordinates of the part of the image that is to map to the top-left corner of the sprite.

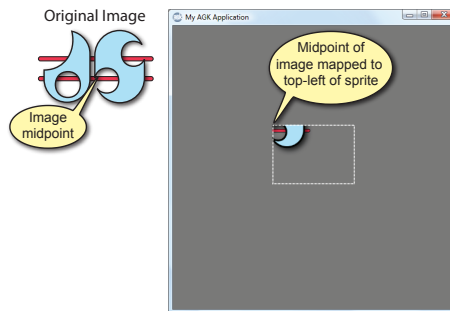
For example, the line

```
SetSpriteUOffset(1,0.5,0.5)
```

would map the middle of the image to the top left corner of the sprite (see FIG-17.32).

FIG-17.32

UV Offset Effects 1



You can also use negative numbers for the *Uoffset* and *Voffset* values. For example, the line

```
SetSpriteUOffset(1,-0.5,-0.5)
```

has the effect shown in FIG-17.33.

FIG-17.33

UV Offset Effects 2

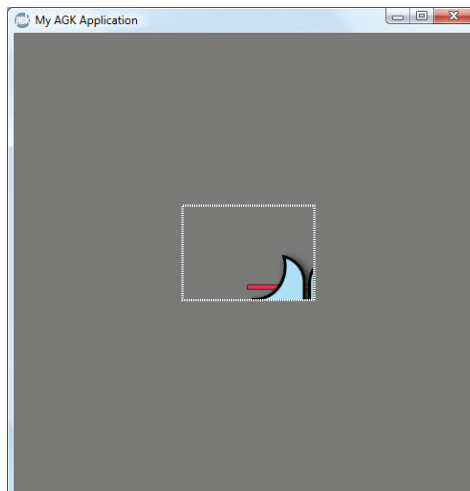
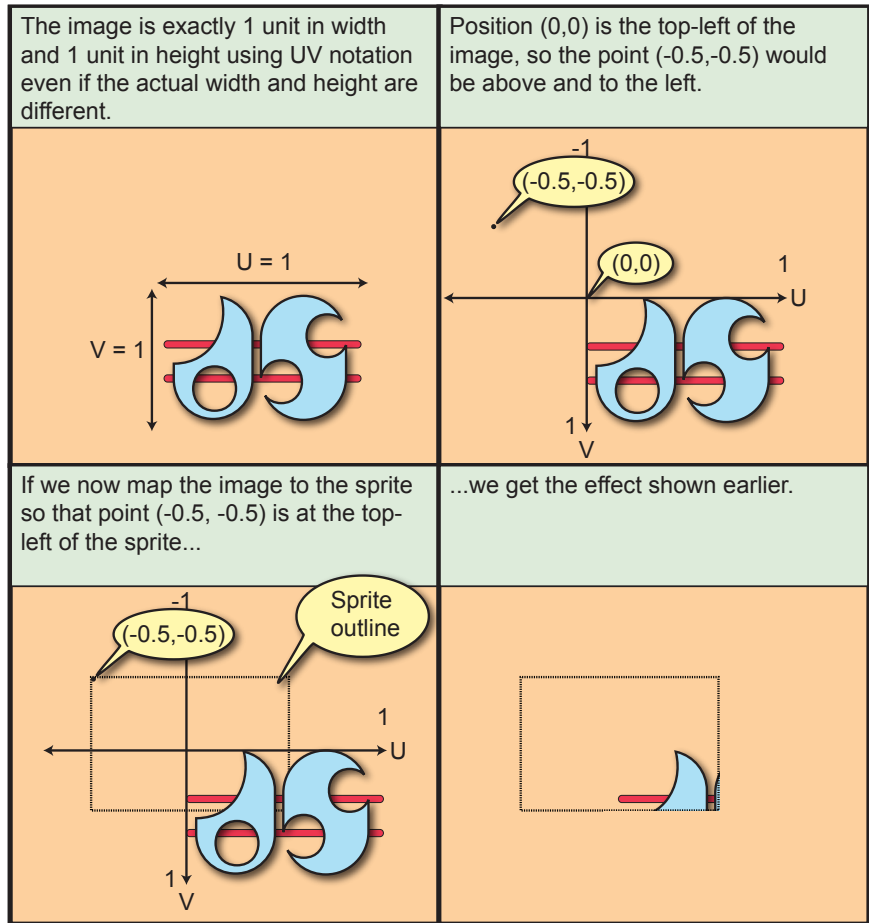


FIG-17.34

UV Offsets Explained



Activity 17.12

Start a new project called *UVOffset* and create a program using image *DS.png* which places the image on a sprite with the UV offset value changing every five seconds.

The following offset values should be used:

0,0 (the default)
 0.25, 0
 0.5, 0.5
 0.7, 0.25
 -0.25, 0
 -0.5, -0.25

Display the offset values used at each stage.

SetImageWrapU() and SetImageWrapV()

The part of the image which falls outside the surface of the sprite is not shown when we use a UV offset, but we can change this using the `SetImageWrapU()` and

`SetImageWrapV()` statements. These cause the image to “wrap round” back onto the sprite’s surface. Using `SetImageWrapU()` causes wrapping on the left and right edges of the sprite; `SetImageWrapV()` wraps the top and bottom edges.

The format of the two statements are shown in FIG-17.35 and FIG-17.36.

FIG-17.35

`SetImageWrapU()`

`SetImageWrapU ((id , iwrap)`

where:

id is an integer value giving the ID of the image. Wrapping will be applied to any sprite using this image.

iwrap is an integer value (0 or 1) which sets horizontal wrapping off (0) or on (1). Zero is the default value.

FIG-17.36

`SetImageWrapV()`

`SetImageWrapV ((id , iwrap)`

where:

id is an integer value giving the ID of the image. Wrapping will be applied to any sprite using this image.

iwrap is an integer value (0 or 1) which sets vertical wrapping off (0) or on (1). Zero is the default value.

Activity 17.13

Modify *UVOffset* by switching on image wrapping in the U direction only and observe changes to the sprite’s image.

Make a second change to the program so that wrapping occurs in both the U and V directions and observe the sprites created.

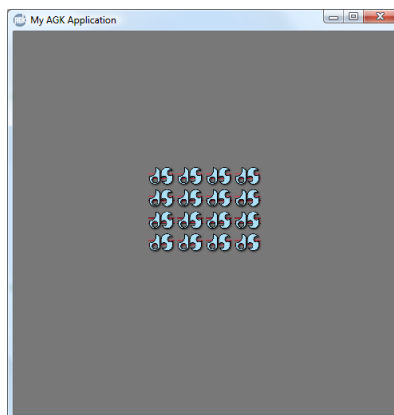
If you also use UV scaling in combination with image wrapping, you will get a repeating image on the sprite. For example, the line

`SetSpriteUVScale(1,0.25,0.25)`

creates the result shown in FIG-17.37.

FIG-17.37

Combine Scaling and Wrapping



The program in FIG-17.38 creates a continually changing UV offset value.

FIG-17.38

Using UV Wrapping

```
rem *** Changing UV Offsets ***

rem *** Grey screen ***
ClearColor(120,120,120)
Sync()
rem *** Load image and allow wrapping ***
LoadImage(1,"DS.png")
SetImageWrapU(1,1)
SetImageWrapV(1,1)
rem *** create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,30,-1)
SetSpritePosition(1,35,35)
rem *** Scale the UV mapping ***
SetSpriteUVScale(1,0.25,0.25)
rem *** Set UV offset value ***
offset# = 0
do
    rem *** Change UV offset ***
    SetSpriteUVOffset(1,offset#,offset#)
    Sync()
    Sleep(100)
    rem *** Change offset Value ***
    offset# = offset# + 0.05
    if offset# > 1
        offset# = 0
    endif
loop
```

Activity 17.14

Start a new project called *ScaledWrap* and implement the code given in FIG-17.38.

What effect is created by the code?

SetSpriteUVBorder()

Sometimes there can be problems when an image is mapped to a sprite. Since the actual number of pixels in an image are unlikely to be an exact match for the number of pixels allocated to a sprite on-screen, there can be a subtle problem with exactly how much of the image will appear on the sprite. This can manifest itself by a small part of the image being missing from the sprite or, when using an atlas image, by “stealing” a pixel from the next image.

AGK handles this by creating a small offset to the mapping of 0.5 pixels. This is known as the **sprite border**.

You can eliminate this offset using `SetSpriteUVBorder()` whose format is shown in FIG-17.39).

FIG-17.39

SetSpriteUVBorder()

`SetSpriteUVBorder (id , iborder)`

where:

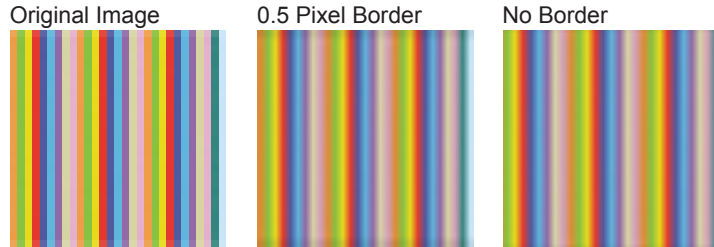
id is the integer value previously assigned as the ID of the sprite.

iborder is an integer value (0 or 1) which leaves the 0.5 pixel offset (0) or removes the offset (1).

FIG-17.40 shows how the border effect changes the mapping of an image to a sprite.

FIG-17.40

The Border Shift Effect



The difference in the mappings is quite a subtle one but you may notice (even in greyscale) that the left and right edges of the two sprite mappings differ slightly. This is also true for the top and bottom edges but is not so obvious from this example.

The program in FIG-17.41 demonstrates the effect of the UV border by displaying a sprite which uses the standard border setting and then, after two seconds, switching to the zero offset option.

FIG-17.41

Using Border Shifting

```
rem *** UV Border ***

LoadImage(1,"Colours.png")
rem *** create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,60,-1)
SetSpritePosition(1,20,20)
rem *** 0.5 UV offset ***
rem *** this is the default ***
rem *** so the statement below ***
rem *** isn't really required! ***
SetSpriteUVBorder(1,0)
Sync()
Sleep(5000)
rem *** Change to no UV offset ***
SetSpriteUVBorder(1,1)
Sync()
do
loop
```

Activity 17.15

Start a new project called *UVBorder* and implement the code given in FIG-17.41. You need to copy *AGKDownloads/Chapter17/Colours.png* to the project's *media* folder.

Observe the effect on the sprite as the border changes. Save your project.

SetSpriteUV()

For ultimate control over how an image is mapped onto a sprite use `SetSpriteUV()`. This allows you to state which part of the image is mapped to each corner of the sprite. The format of the statement is shown in FIG-17.42.

FIG-17.42 SetSpriteUV()

SetSpriteUV ((id , u1 , v1 , u2 , v2 , u3 , v3 , u4 , v4)

where

- id is an integer value giving the ID of the sprite.
- u1,v1 are real values giving the UV coordinates within the image that are to map to the top-left corner of the sprite.
- u2,v2 are real values giving the UV coordinates within the image that are to map to the bottom-left corner of the sprite.
- u3,v3 are real values giving the UV coordinates within the image that are to map to the top-right corner of the sprite.
- u4,v4 are real values giving the UV coordinates within the image that are to map to the bottom-right corner of the sprite.

The program in FIG-17.43 fixes corners 2 to 4 with their normal image to sprite mapping, but chooses a changing random value for the top-left corner.

FIG-17.43

Using SetSpriteUV

```
rem *** Setting a Sprite's UV Mapping ***

rem *** Load image ***
LoadImage(1,"UVTest.png")

rem *** Create a sprite ***
CreateSprite(1,1)
SetSpriteSize(1,50,-1)
SetSpritePosition(1,25,25)

do
  rem *** Create random UV values ***
  u1# = Random(0,1000)/1000.0
  v1# = Random(0,1000)/1000.0

  rem *** Remap the image on the sprite ***
  SetSpriteUV(1,u1#,v1#,0,1,1,0,1,1)
  Sync()
  rem *** Wait 0.5 seconds ***
  Sleep(500)
loop
```

A typical display from the program is shown in FIG-17.44.

FIG-17.44

A Sample Display



Activity 17.16

Start a new project called *SetUV* and implement the code given in FIG-17.43. Copy the file *UVTest.png* to the *media* folder.

Test and save your project.

ResetSpriteUV()

If you have been adjusting the UV mapping of a sprite's image, you can set it back to its default value using `ResetSpriteUV()` (see FIG-17.45).

FIG-17.45

`ResetSpriteUV()`

`ResetSpriteUV (id)`

where

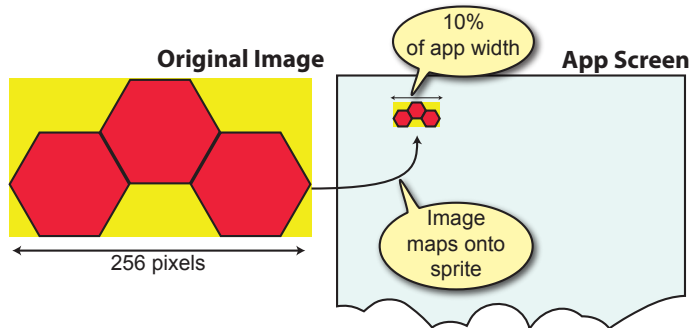
id is an integer value giving the ID of the sprite whose texture mapping is to be reset.

GetSpritePixelFromX()

Normally, the image that is assigned to a sprite will be much larger than the sprite itself. For example, we might place a 256x128 pixel image to a sprite that covers 10% of the width of the app screen. So, in effect, there are 25.6 pixels of the image width squeezed into one tenth of the sprite's width. The idea is visualised in FIG-17.46.

FIG-17.46

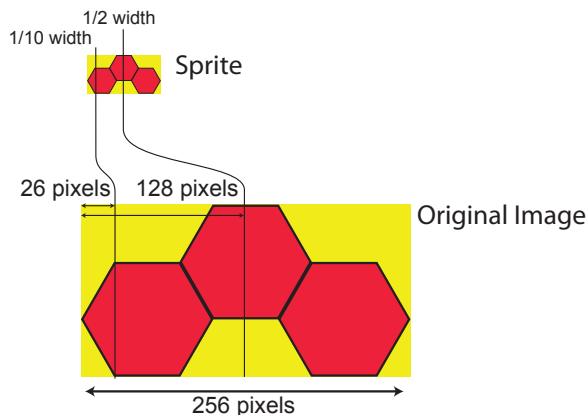
Sprite to Image Mapping 1



So a point one tenth of the way along the sprite would correspond to one tenth of the way along the original image which is (allowing for rounding to the nearest pixel) 26 pixels in from the left (see FIG-17.47).

FIG-17.47

Sprite to Image Mapping 2



The `GetSpritePixelFromX()` statement will return which part of the image maps to a given part of the sprite to which it has been assigned. The statement's format is shown in FIG-17.48).

FIG-17.48

`GetSpritePixelFromX()`

integer `GetSpritePixelFromX` (`id` , `xoffset`)

where:

- id** is the integer value previously assigned as the ID of the sprite.
- xoffset** is a real number giving the distance in from the left edge of the sprite. This distance is measured in percentage or virtual pixels depending on the system being used.

For example, if we create a sprite which is based on a 256 pixel wide image and is 10% the width of the app window, with the lines

```
CreateImage(1,"Hexagons.png")
CreateSprite(1,1)
SetSpriteSize(1,10,-1)
```

then the line

```
Print(GetSpritePixelFromX(1,1))
```

would display the value 26.

For a sprite which is 10% the width of the app window, the second parameter of `GetSpritePixelFromX()` would lie in the range 0 to 10.

The program in FIG-17.49 displays a sprite showing three hexagons. A vertical line shows the position in the sprite being tested and the image pixel equivalent is shown in text form.

FIG-17.49

`GetSpritePixelFromX()`

```
rem *** main image ***
LoadImage(1,"Hexagons.png")
rem *** Vertical line ***
LoadImage(2,"VerticalLine.png")
rem *** Create sprite to be used ***
CreateSprite(1,1)
SetSpriteSize(1,20,-1)
rem *** Create text to show position ***
CreateText(1,"")
SetTextPosition(1,10,10)
rem *** Create vertical line to show position ***
CreateSprite(2,2)
SetSpriteSize(2,0.2,-1)
rem *** Step through sprite ***
for c# = 0 to 20 step 0.5
    rem *** Show pixel equivalent ***
    SetTextString(1,Str(GetSpritePixelFromX(1,c#)))
    rem *** Show position within sprite ***
    SetSpritePosition(2,c#,0)
    Sync()
    Sleep(500)
next c
do
loop
```

Activity 17.17

Start a new project called *SpriteImageMapping* and implement the code given in FIG-17.49. Copy the files *Hexagons.png* and *VerticalLine.png* to the *media* folder. Resize the app window to 500 x 500.

Test and save your project.

GetSpritePixelFromY()

We can do the same sprite-to-original-image conversion, but this time horizontally, using the `GetSpritePixelFromY()` statement (see FIG-17.50).

FIG-17.50

GetSpritePixelFromY() integer `GetSpritePixelFromY` ((`id` , `yoffset`))

where:

- id** is the integer value previously assigned as the ID of the sprite.
- yoffset** is a real number giving the distance in from the top edge of the sprite. This distance is measured in percentage or virtual pixels depending on the system being used.

GetSpriteXFromPixel() and GetSpriteYFromPixel()

As well as finding which pixel is equivalent to a given position on the sprite, we can reverse the process and find which position on the sprite is equivalent to a given position on the image. This is achieved using `GetSpriteXFromPixel()` and `GetSpriteYFromPixel()`. The format for each statement is given in FIG-17.51.

FIG-17.51

GetSpriteXFromPixel() float `GetSpriteXFromPixel` ((`id` , `ipixelx`))
GetSpriteYFromPixel() float `GetSpriteYFromPixel` ((`id` , `ipixelx`))

where:

- id** is the integer value previously assigned as the ID of the sprite.
- ipixelx** is an integer value giving the number of pixels in from the left of the image.
- ipixelx** is an integer value giving the number of pixels down from the top of the image.

SetSpriteSnap()

When a sprite is moving, the theoretical position of the sprite may not map exactly to a pixel. This may cause the sprite to flicker as it adjusts itself to the nearest physical pixel.

FIG-17.52

You can force a sprite to map exactly to a screen pixel using `SetSpriteSnap()` (see FIG-17.52).

SetSpriteSnap() `SetSpriteSnap` ((`id` , `iosp`))

where

- id** is an integer value giving the ID to be assigned to be snapped to the nearest screen pixel.
- iop** is an integer value (0 or 1) specifying if the sprite is to snap to the nearest pixel (1) or not (0).

SetSpriteScissor()

You can perform clipping on a sprite so that any part of the sprite that falls outside a specified rectangular area is not drawn on the screen. This is done using the `SetSpriteScissor()` statement (see FIG-17.53).

FIG-17.53

SetSpriteScissor()

`SetSpriteScissor ((id , x1 , y1 , x2 , y2))`

- id** is an integer value giving the ID of the sprite to be clipped.
- x1,y1** are real values giving the coordinates of the top-left corner of the rectangular area.
- x2,y2** are real values giving the coordinates of the bottom-right corner of the rectangle.

The Sprite Offset Feature

All sprites are created with a fixed point about which they rotate. This point defaults to the exact centre of the sprite. Because most operations on sprites work from the top-left corner of the sprite (for example, `SetSpritePosition()`), this point of rotation is measured relative to that top-left corner and is known as the sprite's **offset**.

SetSpriteOffset()

By using the `SetSpriteOffset()` statement we can change this offset point and make the sprite rotate about a different position. The statement's format is shown in FIG-17.54.

FIG-17.54

SetSpriteOffset()

`SetSpriteOffset ((id , xoffset , yoffset))`

where:

- id** is the integer value previously assigned as the ID of the sprite.
- xoffset, yoffset** are real numbers giving the new position around which the sprite is to rotate. This offset is measured from the top-left corner of the sprite.

We'll see later that other sprite commands also make use of this offset point.

If the rotation point is to be within the body of the sprite, then *xoffset* should lie in the range 0 to `GetSpriteWidth()` and *yoffset* in the range 0 to `GetSpriteHeight()`.

The program in FIG-17.55 displays two arrowed lines on a grid background. The arrows are first rotated about their centre and then about the top-left corner of their sprite.

FIG-17.55

Using SetSpriteOffset()

```

rem *** Sprite Offsets ***

rem *** Grey background ***
SetClearColor(100,100,100)
Sync()

rem *** Background grid image ***
LoadImage(1,"Grid.png")
rem *** Test Image ***
LoadImage(2,"Arrows.png")

rem *** Create grid in background ***
CreateSprite(1,1)
SetSpriteSize(1,100,100)
rem *** Add Sprite ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,40,40)
Sync()
Sleep(1000)

rem *** Rotate sprite about its centre ***
for c = 0 to 360
    SetSpriteAngle(2,c)
    Sync()
next c
Sleep(1000)
Sync()

rem *** Move rotation point to
rem *** top-left corner of sprite ***
SetSpriteOffset(2,0,0)
SetSpritePosition(2,40,40)
Sync()
Sleep(1000)

rem *** Rotate sprite about new point ***
for c = 0 to 360
    SetSpriteAngle(2,c)
    Sync()
next c
do
loop

```

Activity 17.18

Start a new project called *SpriteOffset* and implement the code given in FIG-17.55. The images *Grid.png* and *Arrows.png* must be copied from *AGKDownloads/Chapter17* to the project's *media* folder.

Test the code and observe how the sprite rotates.

Modify the code so that the sprite rotates about its bottom-right corner instead of the top-left corner. Save your project.

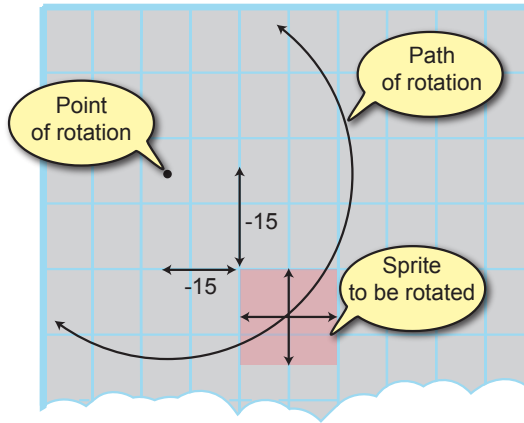
A sprite can be rotated about any point on or off the screen. The only thing to remember is that the offsets given must always be measured from the top-left corner of the sprite being rotated. For example, the line

```
SetSpriteOffset(2,-15,-15)
```

would rotate the arrowed sprite about the point shown in FIG-17.56.

FIG-17.56

Rotation about the Offset
Point



Activity 17.19

Modify *SpriteOffset* so that the arrows rotate about the point shown in FIG-17.56.

Test your code and observe how the sprite rotates. Save your project.

GetSpriteXByOffset() and GetSpriteYByOffset()

You can find the screen coordinates of a sprite's point of rotation using the `GetSpriteXByOffset()` and `GetSpriteYByOffset()`. The format for each of these statements is shown in FIG-17.57.

FIG-17.57

`GetSpriteXByOffset()`
`GetSpriteYByOffset()`

float `GetSpriteXByOffset` (`id`)

float `GetSpriteYByOffset` (`id`)

where:

id is the integer value previously assigned as the sprite's ID.

Activity 17.20

Modify *SpriteOffset* so that the arrows' offset values are displayed in a single text resource before and during each rotation. Save your project.

As you can see from the results of this latest version of the *SpriteOffset* project, the coordinates returned are measured from the top-left of the app screen area.

SetSpritePositionByOffset()

Normally, when we position a sprite, it is the top-left corner of the sprite that is placed at the specified position. However, we can position the sprite using its offset point, rather than the sprite's top-left corner, by calling the `SetSpritePositionByOffset()`

statement (see FIG-17.58 for the statement's format).

FIG-17.58

SetSpritePositionByOffset()

SetSpritePositionByOffset ((id , x , y))

where:

id is the integer value previously assigned as the ID of the sprite.

x,y are real values giving the position on the screen where the sprite is to be positioned.

If we use `SetSpritePositionByOffset()` to place a sprite and, assuming the sprite's offset remains at its default centre position, then it is the centre of the sprite that is placed at the specified point. For example, `SetSpritePositionByOffset(2,80,60)` would move sprite 2 so that its centre was at position (80,60).

The program in FIG-17.59 positions the *Arrows* sprite at (50,50). It then uses `SetSpriteOffset()` to create an offset centred on the middle of the sprite before using `SetSpritePositionByOffset()` to place the middle of the sprite at position (50,50).

FIG-17.59

Using
SetSpritePositionBy
Offset()

```
rem *** Sprite Offset Positioning ***

rem *** Grey background ***
SetClearColor(100,100,100)
Sync()

rem *** Background grid image ***
LoadImage(1,"Grid.png")
rem *** Test Image ***
LoadImage(2,"Arrows.png")

rem *** Create grid in background ***
CreateSprite(1,1)
SetSpriteSize(1,100,100)

rem *** Add Sprite ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
Sync()
Sleep(5000)
SetSpritePositionByOffset(2,50,50)
Sync()
do
loop
```

Activity 17.21

Start a new project called *OffsetPositioning* and implement the code given in FIG-17.59. Copy the necessary files to the *media* folder.

Observe the change of position of the sprite.

What happens if the `SetSpritePositionByOffset()` statement is replaced by a `SetSpritePosition()` statement? Why do we obtain this effect?

SetSpriteScaleByOffset()

We already have a `SetSpriteScale()` statement which allows use to resize a sprite, but this command always keeps the top-left of the sprite as a fixed point as the image changes size. If we would rather keep the sprite's offset point fixed when scaling a sprite then we can use the `SetSpriteScaleByOffset()` statement (see FIG-17.60).

FIG-17.60

`SetSpriteScaleByOffset ((id , xscale , yscale)`

`SetSpriteScaleByOffset()`

where:

- id** is the integer value previously assigned as the ID of the sprite.
- xscale** is a real value giving the scaling factor in the x direction (width). 0.5 will half the width; 2.0 will double it.
- yscale** is a real value giving the scaling factor in the y direction (height).

By default, using this statement will scale towards the middle of the sprite since all sprites have an initial offset value at that position.

The program in FIG-17.61 demonstrates the effects of scaling using first the `SetSpriteScale()` statement (scales to top-left of sprite), then `SetSpriteScaleByOffset()` (scales to centre of sprite). Finally, the offset point is moved to the bottom right of the sprite and `SetSpriteScaleByOffset()` is used again with scaling moves towards that point.

FIG-17.61

Using
`SetSpriteScaleByOffset()`

```
rem *** Sprite Offset Positioning ***
rem *** Grey background ***
SetClearColor(100,100,100)
Sync()
rem *** Background grid image ***
LoadImage(1,"Grid.png")
rem *** Test Image ***
LoadImage(2,"Arrows.png")
rem *** Create grid in background ***
CreateSprite(1,1)
SetSpriteSize(1,100,100)
rem *** Add Sprite ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
Sync()
Sleep(2000)
rem *** Use SetSpriteScale() ****
rem *** Scales towards top-left of sprite ***
rem *** Decrease scale ***
for c# = 1 to .1 step -0.05
    SetSpriteScale(2,c#,c#)
    Sync()
next c#
Sleep(2000)
rem *** Increase scale ***
for c# = .1 to 1.005 step 0.05
    SetSpriteScale(2,c#,c#)
    Sync()
next c#
```



FIG-17.61

(continued)

Using
SetSpriteScaleByOffset()

```

Sleep(2000)

rem *** Use SetSpriteScaleByOffset() ***
rem *** Scales towards centre ***
rem *** Decrease scale ***
for c# = 1 to .1 step -0.05
    SetSpriteScaleByOffset(2,c#,c#)
    Sync()
next c#
Sleep(2000)

rem *** Increase scale ***
for c# = .1 to 1.005 step 0.05
    SetSpriteScaleByOffset(2,c#,c#)
    Sync()
next c#

rem *** Move offset to bottom-right of sprite ***
SetSpriteOffset(2,GetSpriteWidth(2),GetSpriteHeight(2))
SetSpritePosition(2,50,50)
Sleep(2000)

rem *** Reduce scale ***
for c# = 1 to .1 step -0.05
    SetSpriteScaleByOffset(2,c#,c#)
    Sync()
next c#
Sleep(2000)

rem *** Increase scale ***
for c# = .1 to 1.005 step 0.05
    SetSpriteScaleByOffset(2,c#,c#)
    Sync()
next c#
Sync()

do
loop

```

Activity 17.22

Start a new project called *OffsetScaling* and implement the code given in FIG-17.61. Copy the files *Grid.png* and *Arrows.png* to the *media* folder.

Resize the app window to 768 x 1024.

Test and save your project.

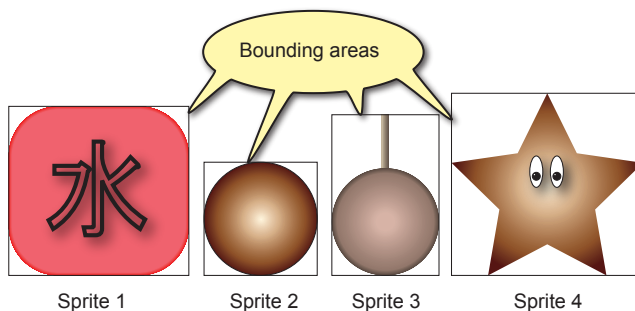
Sprite Bounding Areas

When a sprite is first created, it is surrounded by an invisible **bounding area**. It is this bounding area that is used to determine if sprites have been selected or have collided; the actual visible shape of the sprite is not involved in these calculations.

This initial bounding area is always rectangular in shape for all sprites. The bounding areas of various sprites are made visible in FIG-17.62.

FIG-17.62

Default Bounding Areas



As you can see, a rectangular bounding area is an accurate representation of the sprite only when the sprite image is itself rectangular in nature. An inaccurate bounding area can result in unsatisfactory simulations of various events such as collisions or selections.

SetSpriteShape()

We can change the type of bounding area assigned to a sprite using the `SetSpriteShape()` statement (see FIG-17.63).

FIG-17.63

SetSpriteShape()

```
SetSpriteShape ( id , ishape )
```

where:

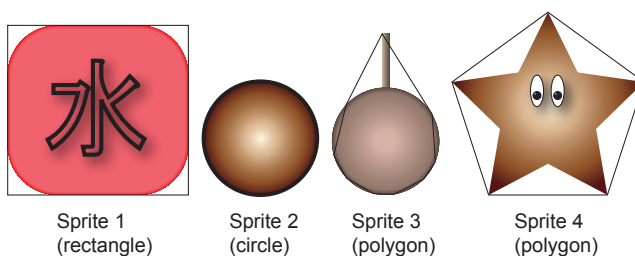
- id** is an integer value giving the ID previously assigned to the sprite.
- ishape** is an integer value (0 to 3) giving the bounding shape to be assigned to the sprite. 0: No bounding area; 1: circle; 2: rectangle; 3: polygon.

Assigning a new bounding area will remove any existing one. If option zero is chosen and no bounding area assigned, then the sprite will not be detected by operations such as selection or collision, although the sprite itself will remain visible.

The new shapes are calculated automatically by AGK. FIG-17.64 shows the new bounding areas for the original four sprites after sprite 2 has been changed to a circle and sprites 3 and 4 to polygons.

FIG-17.64

Sprite Boundary Examples



The circle is a perfect match, and the polygons, although certainly not perfect, are more accurate than the previous rectangle.

The downside to changing the bounding area is that the calculations the AGK engine has to perform when detecting collisions and selections becomes more complex.

SetSpriteShapeBox()

If you wish to create your own bounding area for a sprite, one of the three commands available is `SetSpriteShapeBox()` which allows you to specify your own rectangle.

FIG-17.65

SetSpriteShapeBox()

The format for `SetSpriteShapeBox()` is shown in FIG-17.65.

`SetSpriteShapeBox (id , x1 , y1 , x2 , y2 , fangle)`

where:

- id** is an integer value giving the ID previously assigned to the sprite.
- x1,y1** are a pair of real values giving the coordinates of the top-left corner of the bounding box.
- x2,y2** are a pair of real values giving the coordinates of the bottom-right corner of the bounding box.
- fangle** is a real number giving the angle to which the bounding box is to be rotated. The angle is given in radians.

The positions $(x1,y1)$ and $(x2,y2)$ are measured relative to the sprite's top-left corner.

If we assume we have created a sprite using the commands

```
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,42,10)
```

then we can create our own bounding box using the line:

```
SetSpriteShapeBox(1,0,0,GetSpriteWidth(1),GetSpriteHeight(1),0)
```

This would create exactly the same bounding box as the default one, so the statement would be rather a waste of time! On the other hand, if you wanted to make the bounding box larger than the sprite (perhaps to emulate a force field) then we could use the line

```
SetSpriteShapeBox(1,-1,-1,GetSpriteWidth(1)-1,
↳GetSpriteHeight(1)-1,0)
```

which would create a bounding box exactly 1 unit larger all round (see FIG-17.66).

FIG-17.66

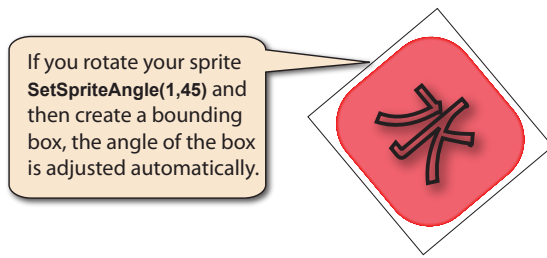
An Expanded Bounding Box



If the sprite is rotated before you add your own bounding box, AGK will automatically rotate your new bounding box too - there is no need for you to supply an angle of rotation when calling `SetSpriteShapeBox()` (see FIG-17.67).

FIG-17.67

The Bounding Box
Adjusts for Rotated
Sprites



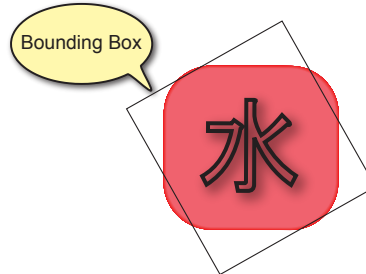
Only if you wish the bounding box to be at an angle to the sprite itself, do you need to supply an angle of other than zero as an argument to `SetSpriteShapeBox()`. For example, the line

```
SetSpriteShapeBox(1,0,0,GetSpriteWidth(1),GetSpriteHeight(1),1.0)
```

would create the setup shown in FIG-17.68.

FIG-17.68

Rotating the Bounding
Box Independently of the
Sprite



SetSpriteShapeCircle()

If you want to create your own bounding circle, use `SetSpriteShapeCircle()` (see FIG-17.69).

FIG-17.69

SetSpriteShapeCircle()

```
SetSpriteShapeCircle ( ( id , x , y , fradius ) )
```

where:

- id** is an integer value giving the ID previously assigned to the sprite.
- x,y** are a pair of real values giving the coordinates of the centre of the bounding circle.
- fradius** is a real number giving the radius of the bounding circle.

Unlike `SetSpriteShapeBox()`, where coordinates are measured from the top-left corner of the sprite, the coordinates here are measured from the sprite's current offset position (which by default is at the centre of the sprite), so, assuming you want the circle to share the same centre as the sprite, `x` and `y` would both have a value of zero.

SetSpriteShapePolygon()

FIG-17.70

SetSpriteShapePolygon()

The final option for creating your own bounding area is to create a bounding polygon using the `SetSpriteShapePolygon()` statement (see FIG-17.70).

```
SetSpriteShapePolygon ( ( id , icount , index , x , y ) )
```

where:

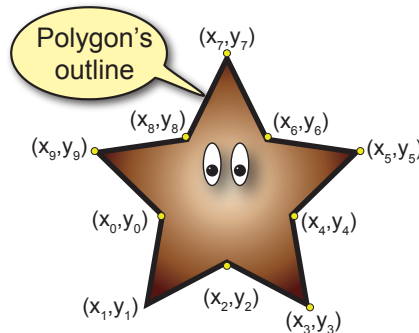
- id** is an integer value giving the ID previously assigned to the sprite.
- icount** is an integer value (3 to 12) giving the number of vertices in the polygon.
- index** is an integer value giving the index of this specific vertex. The first vertex has an index setting of zero.
- x,y** are a pair of real values giving the coordinates of the vertex. The coordinates are measured from the sprite's offset.

The statement has to be called once for each vertex in the polygon.

If we assume we want to set up a polygon for the shape shown in FIG-17.71, then we would need to call `SetSpriteShapePolygon()` ten times - once for each vertex.

FIG-17.71

Identifying the Vertices on a Polygon



Since it would be impractical to figure out the coordinates of the polygon's vertices manually, a simple example of how you might create a program to do this is shown in FIG-17.72.

FIG-17.72

Using
`SetSpriteShapePolygon()`

```
rem *** Set Points for a Bounding Polygon ***

rem *** Main Variables ***
spritewidth as float
spriteheight as float
spriteCentreX as float
spriteCentreY as float
dim vertices[24] as float

rem *** Load images ***
LoadImage(3, "Point.png")
LoadImage(2, "Finished.png")
LoadImage(1, "Sprite4.png")
rem *** Set background colour ***
SetClearColor(120,120,120)
Sync()

rem *** Create sprites ***
CreateSprite(1,1)
SetSpriteSize(1,50,-1)
SetSpritePosition(1,5,25)
```



FIG-17.72

(continued)

Using
SetSpriteShapePolygon()

```

CreateSprite(2,2)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,80,2)
SetSpriteDepth(2,9)
CreateSprite(3,3)
SetSpriteSize(3,3,-1)

rem *** Assign main variables values ***
spritewidth = GetSpriteWidth(1)
spriteheight = GetSpriteHeight(1)
spriteCentreX = GetSpriteXByOffset(1)
spriteCentreY = GetSpriteYByOffset(1)

rem *** Set next sprite number to be used ***
nextSprite = 4
rem *** All points entered flag ***
done = 0
rem *** Count of entries in array ***
count = 0

rem *** Mark each apex ***
repeat
    rem *** Move vertex marker to pointer position ***
    SetSpritePositionByOffset(3,GetPointerX(),GetPointerY())
    rem *** If pointer pressed ***
    if GetPointerPressed() = 1
        rem *** If over button, process is complete ***
        if GetSpriteHit(GetPointerX(),GetPointerY())=2
            done = 1
        else
            rem *** Clone vertex marker sprite ***
            CloneSprite(nextsprite,3)
            rem *** Save position of marker ***
            vertices[count] = GetPointerX()-spriteCentreX
            vertices[count+1] = GetPointerY()-spriteCentreY
            rem *** Add to
            count = count + 2
            inc nextsprite
        endif
    endif
    Sync()
until done = 1

rem *** Construct the bounding polygon ***
vcount = count / 2
for c = 0 to vcount-1
    SetSpriteShapePolygon(1,vcount, c,vertices[c*2],
        ↵vertices[c*2+1])
next c

rem *** Use physics option to show bounding polygon ***
SetSpritePhysicsOn(1,2)
SetPhysicsDebugOn()
SetPhysicsGravity(0,0)

do
    Sync()
loop

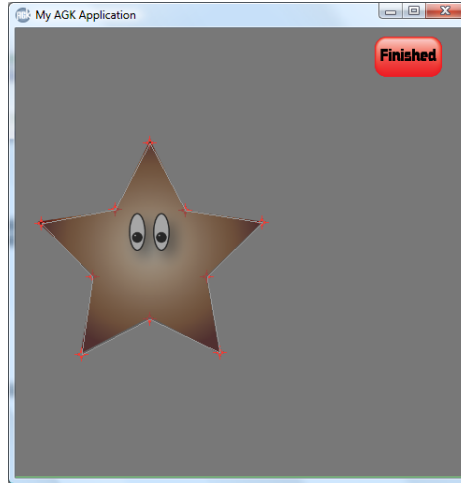
```

The program shows the sprite for which the bounding polygon is to be produced. You

can then click on each of the sprite's vertices, leaving a marker sprite at each point. When all the vertices have been marked, pressing the button in the top-right corner will end the process and then display the new bounding polygon around the sprite. A snapshot from the program is shown in FIG-17.73.

FIG-17.73

Program Display



In order to display the outline of the sprite, the program in FIG-17.72 makes use of three physics statements near the end of the code. These statements are covered in Chapter 20.

Activity 17.23

Start a new project called *BoundingPolygon* and implement the code given in FIG-17.72. Copy the files *Point.png*, *Finished.png* and *Sprite4.png* into the project's *media* folder. Test and save your project.

What line of code positions the marker sprites when the mouse button is clicked?

Where are the coordinates for each vertex stored? Which lines of code set up the bounding polygon?

Sprite Groups

SetSpriteGroup()

Sprites can be grouped. Each sprite belonging to the same group is given an identical group ID. For example, if we were writing a chess program, we might want all of the black pieces to belong to one group and the white pieces to a second group. Grouping can be useful since we can then control the detection of hits and collisions according to these groups.

You can assign a sprite to a specific group using the `SetSpriteGroup()` statement (see FIG-17.74).

FIG-17.74

`SetSpriteGroup()`

`SetSpriteGroup (id , igroup)`

where

id

is an integer value giving the ID of the sprite.

igroup is an integer value representing the group to which the sprite is to be assigned

By default, all sprites are assigned to group zero.

Grouping sprites also affects how they behave during collisions.

GetSpriteGroup()

To determine which group a sprite has been assigned to, use `GetSpriteGroup()` (see FIG-17.75).

FIG-17.75

`GetSpriteGroup()`

integer `GetSpriteGroup` (`id`)

where

id is an integer value giving the ID of the sprite.

The function will return the group to which the sprite has been assigned.

GetSpriteHitGroup()

When sprites have been grouped, we have the option to detect sprite hits belonging to a specific group only. For example, if we have displayed a menu which has been implemented using several grouped sprites, we might want to ignore any hits on sprites outside this group until that menu has been closed.

To check for hits in a specific sprite group, use `GetSpriteHitGroup()` (see FIG-17.76).

FIG-17.76

`GetSpriteHitGroup()`

integer `GetSpriteHitGroup` (`igroup` , `x` , `y`)

where

igroup is an integer value representing the group to be checked.

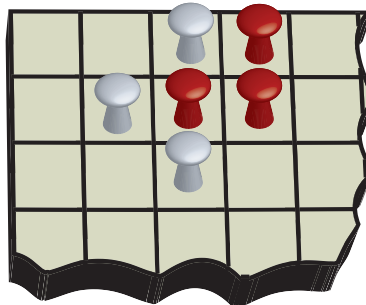
x,y are real values giving the coordinates of the point to be checked for a hit.

If position (x,y) lies on a sprite in the specified group, the ID of that sprite is returned by the function, otherwise zero is returned.

FIG-17.77 shows part of an imaginary board game. The dark pieces in the game have been assigned to group 1 and the white pieces to group 2; the board itself remains in the default group 0.

FIG-17.77

Grouping Playing Pieces



The player moves a piece by clicking on an appropriate sprite. As you can see, with the pieces in such close proximity to each other, it would be easy for a player to touch an opponent's piece. We can use grouping to avoid any problems. The program in FIG-17.78 demonstrates piece selection using groups (note that the piece selected is highlighted but doesn't move).

FIG-17.78

Grouped Playing Pieces

```
rem *** Using Sprite Groups to ***
rem *** Limit User Selection ***

rem *** Load images ***
LoadImage(1,"Board.png")
LoadImage(2,"PieceRed.png")
LoadImage(3,"PieceWhite.png")

rem *** Create board ***
CreateSprite(1,1)
SetSpriteSize(1,100,-1)

rem *** Create red sprites ***
CreateSprite(2,2)
SetSpriteSize(2,8,-1)
SetSpritePosition(2,27,23)
CloneSprite(3,2)
SetSpritePosition(3,39,23)
CloneSprite(4,2)
SetSpritePosition(4,39,15)
rem *** Place in group 1 ***
for c = 2 to 4
    SetSpriteGroup(c,1)
next c

rem *** Create white sprites ***
CreateSprite(5,3)
SetSpriteSize(5,8,-1)
SetSpritePosition(5,27,15)
CloneSprite(6,5)
SetSpritePosition(6,15,23)
CloneSprite(7,5)
SetSpritePosition(7,27,31)
rem *** Place in group 2 ***
for c = 5 to 7
    SetSpriteGroup(c,2)
next c

rem *** Set active group ***
activegroup = 1
rem *** Set pressed state ***
pressed = 0
do
    rem *** If changes to pressed ***
    if GetPointerState()=1 and pressed = 0
        rem *** Record as pressed ***
        pressed = 1
        rem *** Get ID of any sprite from the correct group ***
        rem *** that has been hit ***
        id = GetSpriteHitGroup(activegroup,GetPointerX(),
            ↵GetPointerY())
        rem *** If a sprite was selected ***
        if id <> 0
            rem *** Change its colour ... ***
```



FIG-17.78

(continued)

Grouped Playing Pieces

```

        SetSpriteColorRed(id,0)
        Sync()
        Sleep(500)
        rem *** ... then change it back ***
        SetSpriteColorRed(id,255)
        Sync()
        rem *** Change to other sprite group ***
        activegroup = 3 - activegroup
    endif
else
    rem *** Record as not pressed ***
    pressed = 0
endif
Sync()
loop

```

Activity 17.24

Start a new project called *ControlledHits* and implement the code given in FIG-17.78. Copy the files *Board.png*, *PieceRed.png* and *PieceWhite.png* into the project's *media* folder. Set the app window size to 768 x 1024.

Check that pieces must be selected in a red-white order and that the board itself cannot be selected. Save your project.

SetSpriteCategoryBits()

As well as belonging to a group, a sprite can also be assigned to one or more of 16 categories. We can think of these as alternate groupings or sub-groupings. We can use these categories to gain greater control over which sprites can be hit.

To set the categories to which a sprite is to belong, use `SetSpriteCategoryBits()` (see FIG-17.79).

FIG-17.79`SetSpriteCategoryBits()`

where

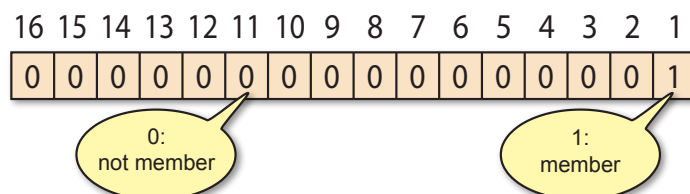
id is an integer value giving the ID of the sprite.

icats is an integer value representing the categories to which a sprite is to belong.

The categories are stored as a 16 bit value (see FIG-17.80) and hence, *icats* is normally stated as a hexadecimal or binary number.

FIG-17.80

How Sprite Categories are Recorded

Category Bits

By default, every sprite is a member of category 1.

To make sprite 1 a member of categories 13, 12, 5, 4, 3 and 2 we would use either of the following statements:

```
SetSpriteCategoryBits(1,0x181E) // Hexadecimal
SetSpriteCategoryBits(1,%0001100000011110) //Binary
```

A sprite's categories can be changed at any time.

SetSpriteCategoryBit()

If you want to modify just a single bit in a sprite's category value without changing the other category settings, then you can use `SetSpriteCategoryBit()` (see FIG-17.81).

FIG-17.81

SetSpriteCategoryBit()

`SetSpriteCategoryBit` ((`id` , `index` , `iflag`))

where

- id** is an integer value giving the ID of the sprite.
- index** is an integer value (1 to 16) giving the category whose value is to be set.
- iflag** is an integer value (0 or 1) giving the category setting.
(0: not member ; 1: member).

For example, we could have sprite 1 become a member of category 9 using the line

```
SetSpriteCategoryBit(1,9,1)
```

GetSpriteHitCategory()

Just as we could use groups to control sprite hits, so we can also use categories in the same way allowing hits only for sprites in specified categories. This is done using the `GetSpriteHitCategory()` statement (see FIG-17.82).

FIG-17.82

GetSpriteHitCategory()

integer `GetSpriteHitCategory` ((`icats` , `x` , `y`))

where

- icats** is an integer value representing the group to be checked.
- x,y** are real values giving the coordinates of the point to be checked for a hit.

If position (x,y) lies on a sprite belonging to any of the specified categories, the ID of that sprite is returned by the function, otherwise zero is returned.

The program in FIG-17.83 is a variation on the previous program. In this version, categories rather than groups are used to identify the pieces. In addition, two pieces are placed off the board and therefore cannot be selected.

```

rem *** Using Sprite Categories to ***
rem *** Limit User Selection ***

rem *** Categories ***
rem *** 1 - all sprites                (bit value 1)
rem *** 2 - red piece on board        (bit value 2)
rem *** 3 - white piece on board      (bit value 4)
rem *** 4 - any piece off board       (bit value 8)

rem *** Load images ***
LoadImage(1,"Board.png")
LoadImage(2,"PieceRed.png")
LoadImage(3,"PieceWhite.png")

rem *** Create board ***
CreateSprite(1,1)
SetSpriteSize(1,100,-1)

rem *** Create red sprites ***
CreateSprite(2,2)
SetSpriteSize(2,8,-1)
SetSpritePosition(2,27,23)
CloneSprite(3,2)
SetSpritePosition(3,39,23)
CloneSprite(4,2)
SetSpritePosition(4,39,75)
rem *** Place in category 2 ***
for c = 2 to 4
    SetSpriteCategoryBits(c,0X0002)
next c
rem *** Sprite 4 is off-board, remove from cat 2 ***
SetSpriteCategoryBit(4,2,0)
SetSpriteCategoryBit(4,4,1) //Off-board

rem *** Create white sprites ***
CreateSprite(5,3)
SetSpriteSize(5,8,-1)
SetSpritePosition(5,27,15)
CloneSprite(6,5)
SetSpritePosition(6,15,23)
CloneSprite(7,5)
SetSpritePosition(7,27,75)
rem *** Place in category 3 ***
for c = 5 to 7
    SetSpriteCategoryBits(c,0X0004)
next c
rem *** Sprite 7 off board, remove from cat 3 ***
SetSpriteCategoryBit(7,3,0)
SetSpriteCategoryBit(7,4,1) //Off-board

rem *** Set active category ***
activecategory = 2 //On-board red
rem *** Set pressed state ***
pressed = 0
do
    rem *** If changes to pressed ***
    if GetPointerState()=1 and pressed = 0

```



FIG-17.83

(continued)

Using Categories

```

rem *** Record as pressed ***
pressed = 1
rem *** Get ID of any sprite from the correct group ***
rem *** that has been hit ***
id = GetSpriteHitCategory(activecategory,GetPointerX(),
↳GetPointerY())
rem *** If a sprite was selected ***
if id <> 0
    rem *** Change its colour ... ***
    SetSpriteColorRed(id,0)
    Sync()
    Sleep(500)
    rem *** ... then change it back ***
    SetSpriteColorRed(id,255)
    Sync()
    rem *** Change to other sprite category ***
    activecategory = 6 - activecategory //Swap
    ↳between red and white onboard
endif
else
    rem *** Record as not pressed ***
    pressed = 0
endif
Sync()
loop

```

Activity 17.25

Modify *ControlledHits* to match the code given in FIG-17.83 and check that the off board pieces cannot be selected.

Change the code so that off board pieces can always be selected, irrespective of which player is to move (red and white pieces on board should still alternate).

Save your project.

Sprite groups and categories can also be used to control collisions, as we will see later.

Moving Sprites

In many games we will want at least some of the sprites being displayed to move about the screen. Perhaps such a sprite represents a ball, a missile, or a character.

A moving character has velocity. That is to say, it has speed and a direction in which it moves. When movement is either horizontal or vertical, then the data required is trivial. We need only set a speed with a line such as

```
speed = 1
```

and then move the sprite by this amount on each iteration:

```

do
    SetSpritePosition(id,GetSpriteX(id)+speed,GetSpriteY(id))
    Sync()
loop

```

The program in FIG-17.84 demonstrates the effect achieved by using this code on a ball sprite.

FIG-17.84

Moving a Sprite
Horizontally

```
rem *** Moving a sprite ***

rem *** Load image ***
LoadImage(1,"Ball.png")

rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)

rem *** Set Speed ***
speed = 1

rem *** Move sprite ***
do
    SetSpritePosition(id,GetSpriteX(id)+speed, GetSpriteY(id))
    Sync()
loop
```

Activity 17.26

Start a new project called *MovingBall* and implement the code given in FIG-17.84. (Remember to copy *AGKDownloads/Chapter17/Ball.png* to the project's *media* folder.)

Run the program. In which direction does the ball travel?

Modify the program so that the ball travels in the opposite direction.

Modify the program again so that the ball travels vertically down and change the speed of the ball to 2.

Save your project.

When the ball travels in a direction which is at an angle to the vertical and horizontal, then things get a little more complicated (see FIG-17.85).

FIG-17.85

Calculating a Sprite's *x*
and *y* Offsets from its
Velocity

The diagram shown here makes use of conventional math layout which assumes that the positive *y*-axis points in an upward direction and that angles are measured in a counterclockwise direction. On a computer screen, the positive *y*-axis points downwards and angles are measured in a clockwise direction.

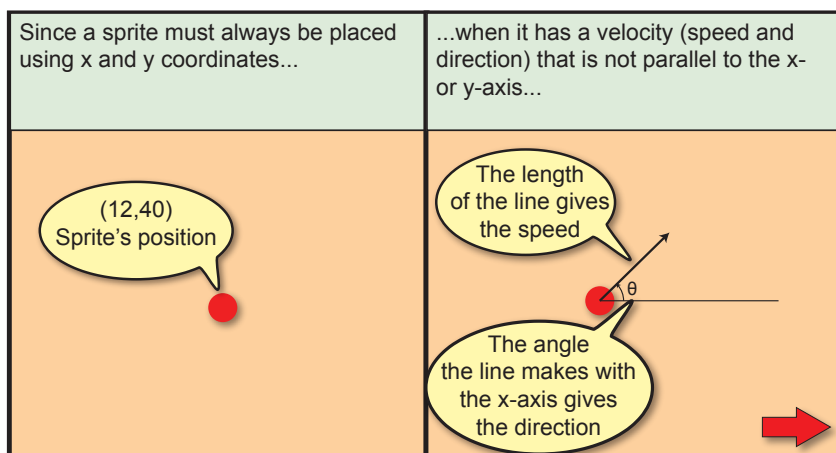
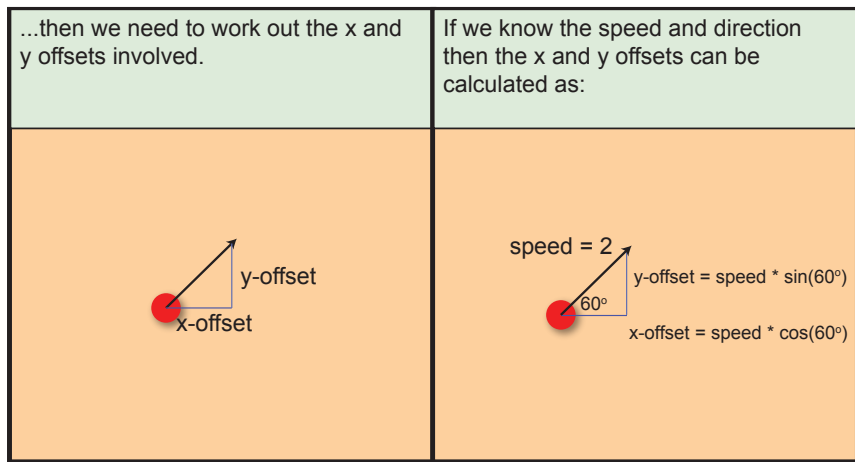


FIG-17.85

(continued)

Calculating a Sprite's x and y Offsets from its Velocity

**Activity 17.27**

Modify *MovingBall* so that the ball's velocity has a speed of 2% at an angle of 60° .

Test and save your project.

In practice, it is more likely that we will simply calculate a value for the x and y offsets directly rather than the speed and angle.

As you have seen when running the *MovingBall* program, a speed of 2 is fairly fast, so typical offset values would be calculated with lines such as:

```
xoffset# = Random(0,10)/5.0
yoffset# = Random(0,10)/5.0
```

which will give offsets in the range 0.0 to 2.0 in steps of 0.2.

Values in this range will always result in a left-downward movement, since both offsets will always be positive. To achieve values in the range -2 to +2 (thereby allowing movement in any direction, we could use the lines:

```
xoffset# = Random(0,20)/5.0 -2
yoffset# = Random(0,20)/5.0 -2
```

Activity 17.28

Modify *MovingBall* so that the x and y offsets are randomly generated and lie in the range -2 to +2 with a step size of 0.1.

Run your program several times to check that the direction of the ball varies.

Save your program.

In some games, when a sprite reaches the edge of the screen, it simply reappears at the same position on the opposite edge. For example, we could check if our ball sprite has reached the right-hand edge of the app window using the line:

```
if GetSpriteX(id) >= 100
```

When the condition is true we need to reset the x-coordinate of the sprite to zero, leaving the y value unchanged:

```
if GetSpriteX(id) >= 100
    SetSpritePosition(id,0,GetSpriteY(id))
```

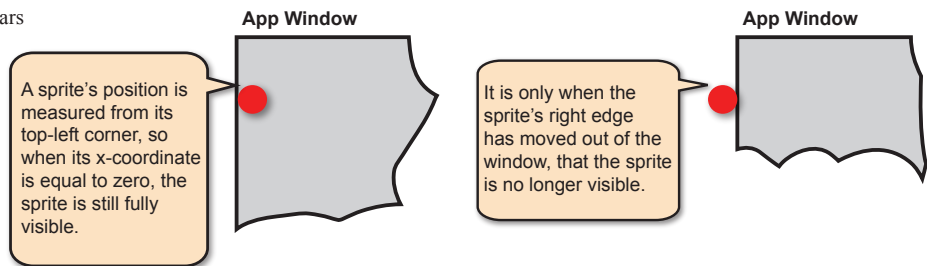
For a moment, you might be tempted to assume that we can check if the sprite has moved off the left edge of the screen using the line

```
if GetSpriteX(id) <= 0
```

but, in fact, the sprite will still be on-screen when its x-coordinate is equal to zero (see FIG-17.86).

FIG-17.86

When a Sprite Disappears from the Screen



If we know the width of the sprite, we can use that to modify the condition in the `if` statement. Since we created the ball to have a width of 5%, then the ball will be off the left edge of the screen when the statement

```
if GetSpriteX(id) <= -5
```

is true, so we can bring the sprite back in on the right edge with the lines:

```
if GetSpriteX(id) <= -5
    SetSpritePosition(id,100,GetSpriteY(id))
```

To achieve the same effect for the top and bottom edges, the tests are

```
if GetSpriteY(id) >= 100 // bottom edge
```

and

```
if GetSpriteY(id) <= -GetSpriteHeight(id) // top edge
```

Note that, to check if the sprite has moved off the top edge, we needed to get the sprite's height. Although we set the width of the sprite to 5 when it was created, the height value was given as -1, allowing AGK to calculate the sprite's height in such a way as to maintain the correct width to height ratio, so we cannot be sure of the exact height of the sprite without calling the `GetSpriteHeight()` function.

In the final code, it's best if we don't allow two off-edge moves to be reset during the same iteration, since this can lead to a deadlock situation in a few, rare circumstances. So, the final code for resetting off-edge moves is:

```
if GetSpriteX(id) >= 100
    SetSpritePosition(id,0,GetSpriteY(id))
elseif GetSpriteX(id) <= -5
```



```

        SetSpritePosition(id,100,GetSpriteY(id))
    elseif GetSpriteY(id) >= 100
        SetSpritePosition(id,GetSpriteX(id),0)
    elseif GetSpriteY(id) <= -GetSpriteHeight(id)
        SetSpritePosition(id,GetSpriteX(id),100)
    endif
end

```

Activity 17.29

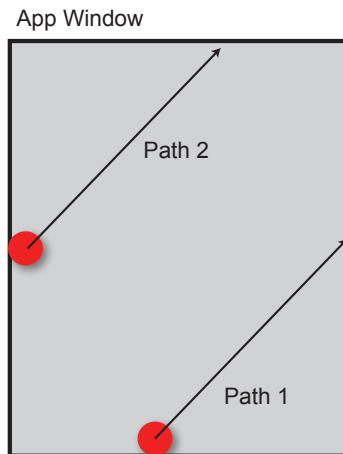
Modify *MovingBall* so that the ball reappears at the opposite edge when it leaves the screen.

Run the program several times before saving your project.

You may have noticed that the overall effect does not seem quite right for some trajectories (see FIG-17.87).

FIG-17.87

A Typical Sprite Path



When the sprite follows *Path 1*, it disappears off the left edge and re-enters along *Path 2*. And, although this might seem logical when we came up with the code for the edge checks, in practice it doesn't look natural.

A better way to think of the app window is as an unwrapped sphere. If we imagine an object orbiting a sphere, then if we watch it appear over one horizon and disappear off another, some time later we would expect it to reappear at its original position over the first horizon. Relating this idea to FIG-17.87, if the ball follows Path 1, we would expect it to leave the left edge and then reappear at the same spot at the bottom of the screen and follow exactly the same path (Path 1).

Since the path of the ball is a straight line, this is where another bit of the math you ignored at school is actually useful. The equation of any straight line is given as:

$$y = mx + c$$

where

x and y represent the coordinates of a point on the line

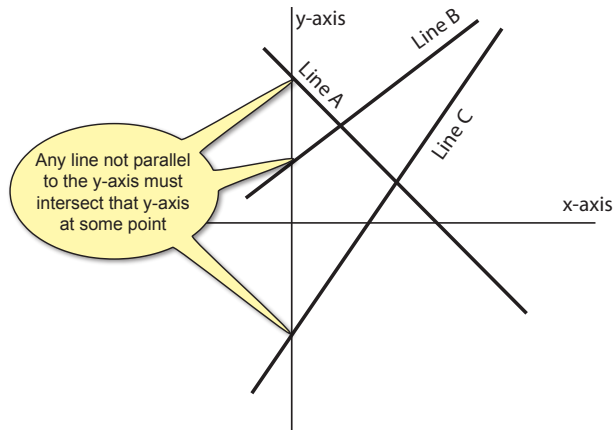
m is the gradient of the line

c is the point at which the line intersects the y -axis

FIG-17.88 shows three examples of lines.

FIG-17.88

Lines



If we want the ball sprite in our program to follow the same path as it disappears off an edge of the screen and reappears on another, then we need to know the equation of the line the ball is following.

We can work out the gradient of the line easily, because this is just the *yoffset#* value divided by the *xoffset#* value:

```
m# = yoffset#/xoffset#
```

However, there is a problem. If a line is parallel to the y-axis, the *xoffset#* value would be zero and the line would have an infinite gradient - a value that the computer cannot handle. So, when we code the calculation, we must take this possibility into account, which we can do with the lines:

```
if xoffset# = 0
  m# = 999999
else
  m# = yoffset#/xoffset#
endif
```

Notice that we have set the value of *m#* to a very large (but not infinite) number when the line is parallel to the y-axis. This will be a good enough approximation for our line.

Now we have to calculate the value of *c*. We can do this if we know at least one point that lies on the line. Luckily, we do. In the code

```
SetSpritePosition(id,47.5,47.5)
```

we started the ball at position (47.5,47.5) so we know that this point lies on the trajectory of our line.

If

$$y = mx + c$$

then it follows that

$$c = y - mx$$

So we can calculate *c#* in our program using the line

```
c# = 47.5 - m#*47.5
```

When the sprite disappears off the right edge of the screen ($x \geq 100$) and we want to bring it back on at the left edge of the screen ($x = 0$) then the value of y at that point is

```
y = m#*0 + c#
```

which, of course, is simply

```
y = c#
```

and when the ball moves off the left edge ($x \leq -\text{ball's width}$) and returns at the right edge, the value for y this time is

```
y = m#*100+c#
```

If the ball moves off the bottom edge ($y \geq 100$) and reappears at the top, then, this time, it is the value of x we need to calculate. Again starting with

```
y = mx + c
```

it follows that

```
x = (y-c)/m
```

so, when the ball reappears at the top of the screen $y = 0$, meaning that

```
x = -c#/m#
```

and when the ball reappears at the bottom of the screen (where $y = 100$)

```
x = 100-c#/m#
```

The updated version of *MovingBall* is shown in FIG-17.89.

FIG-17.89

Maintaining a Sprite's Trajectory

```
rem *** Moving a sprite ***

rem *** Load image ***
LoadImage(1,"Ball.png")

rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)

rem *** Set velocity ***
xoffset# = Random(0,40)/10.0 -2
yoffset# = Random(0,40)/10.0 -2

rem *** Calculate gradient ***
if xoffset# = 0
    m# = 999999
else
    m# = yoffset#/xoffset#
endif

rem *** Calculate intersection with y-axis ***
c# = 47.5 - m# * 47.5
```



FIG-17.89

(continued)

Maintaining a Sprite's
Trajectory

```

rem *** Move the sprite ***
do
    SetSpritePosition(id,GetSpriteX(id)+xoffset#,
    ↪GetSpriteY(id)+yoffset#)
    if GetSpriteX(id) >= 100
        SetSpritePosition(id,0,c#)
    elseif GetSpriteX(id) <= -5
        SetSpritePosition(id,100,m#*100+c#)
    elseif GetSpriteY(id) >= 100
        SetSpritePosition(id,-c#/m#,0)
    elseif GetSpriteY(id) <= -GetSpriteHeight(id)
        SetSpritePosition(id,(100-c#)/m#,100)
    endif
    Sync()
loop

```

Activity 17.30

Modify *MovingBall* to match the code given in FIG-17.89.

Run the program several times and check that the ball follows the same path as it moves across the screen.

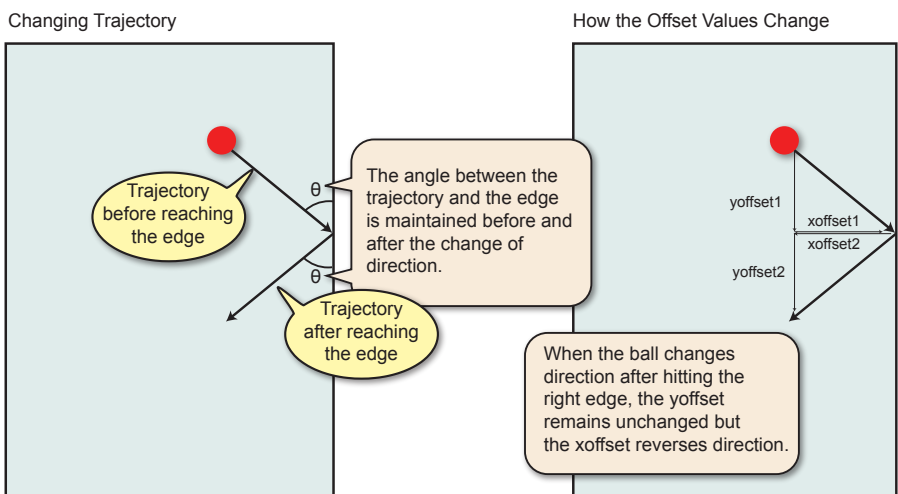
Save your project.

Not all programs allow a sprite to move off-screen. Some treat the the edges of the screen or window as a barrier which approaching sprites simply “bounce off”.

To achieve this effect, we need only negate either the *xoffset* or *yoffset* value. Which one is negated depends on which edge is reached. When the left or right edges are reached, it is the *xoffset* that must be changed (see FIG-17.90); for the top and bottom edges, the *yoffset* must be changed.

FIG-17.90

Sprite Bounce



The code for this version of *MovingBall* is given in FIG-17.91.

FIG-17.91

Implementing Sprite
Bounce

```

rem *** Moving a sprite ***

rem *** Load image ***
LoadImage(1,"Ball.png")

rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)

rem *** Set velocity ***
xoffset# = Random(0,40)/10.0 -2
yoffset# = Random(0,40)/10.0 -2

do
    rem *** Place ball ***
    SetSpritePosition(id,GetSpriteX(id)+xoffset#,
        ↵GetSpriteY(id)+yoffset#)
    rem *** If ball hits left or right edge, negate xoffset ***
    if GetSpriteX(id) >= (100-GetSpriteWidth(id)) or
        ↵GetSpriteX(id) <= 0
        xoffset# = -xoffset#
    rem *** If ball hits top or bottom edge, negate yoffset ***
    elseif GetSpriteY(id) >= (100-GetSpriteHeight(id)) or
        ↵GetSpriteY(id) <= 0
        yoffset# = -yoffset#
    endif
    Sync()
loop

```

Activity 17.31

Start a new project named *MovingBall2* and implement the code given in FIG-17.91. Test and save your project.

In the code, you can see that as an edge is reached either the *xoffset#* or *yoffset#* variable is negated making the ball move in a new direction. Also note that the boundary values are changed slightly from the previous version of *MovingBall* since we don't want the ball to move off-screen before changing direction.

GetSpriteCollision()

With bounding areas in place around each sprite, AGK makes use of these to detect when one sprite comes into contact with another. In a game context, such a collision may represent something as simple as a ball hitting a bat or a missile hitting a spaceship.

The `GetSpriteCollision()` statement returns 1 when the bounding areas of two specified sprites overlap. The statement has the format shown in FIG-17.92.

FIG-17.92

GetSpriteCollision()

integer `GetSpriteCollision` (`id1` , `id2`)

where:

id1 is an integer value giving the ID of the first sprite.

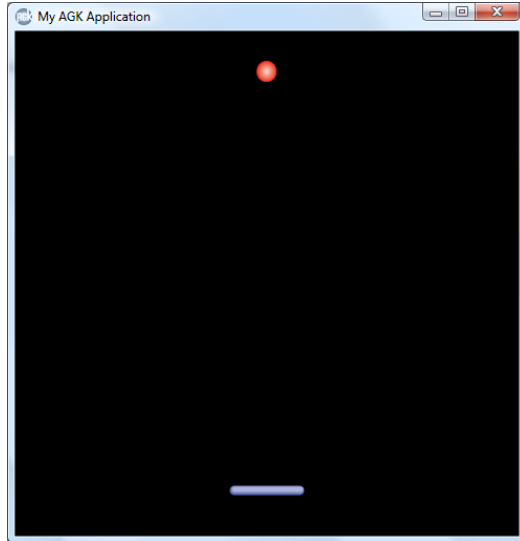
id2 is an integer value giving the ID of the second sprite.

If the sprites overlap, the function returns 1, otherwise zero is returned.

The next program demonstrates collision detection by dropping a ball onto a bat and making the ball bounce back when a collision is detected. The setup for the demonstration is shown in FIG-17.93.

FIG-17.93

Bat and Ball Program
Display



The program code is shown in FIG-17.94.

FIG-17.94

Bat and Ball Program
Code

```
rem *** Bat and ball ***

rem *** Load images ***
LoadImage(1,"Ball.png")
LoadImage(2,"Bat.png")

rem *** Create Sprites ***
CreateSprite(1,1)
SetSpriteSize(1,4,-1)
SetSpritePosition(1,48,5)
CreateSprite(2,2)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,42.5,90)
Sync()

rem *** Wait for 2 seconds ***
Sleep(2000)

rem *** Set ball's speed in y direction ***
bally = 1

do
    rem *** if ball hits bat ***
    if GetSpriteCollision(1,2)
        rem *** Reverse ball's direction ***
        bally = - bally
    endif
    rem *** Redraw ball's position ***
    SetSpritePosition(1,GetSpriteX(1),GetSpriteY(1)+bally)
    Sync()
loop
```

Activity 17.32

Start a new project called *BatandBall* and implement the code given in FIG-17.94 copying *Ball.png* and *Bat.png* from *AGKDownloads/Chapter17* to the *media* folder.

Test and save your project.

Activity 17.33

Modify *BatandBall* so that the ball starts at the top of the screen with a random velocity (make the *yoffset* lie in the range 0.5 to 2.0 and the *xoffset* lie in the range -2 to +2, both in increments of 0.1).

If the ball hits the left, right, or top edges, make it bounce off the edge, but if it reaches the bottom edge, the game is over.

Create a physical joystick option to allow the bat to be moved left or right. If you are running this on a standard PC without a physical joystick, you can use the *A* and *D* keys to move the bat.

If you are running the program on your tablet or phone, a virtual joystick will appear and this can be used to control the bat.

Using the virtual joystick on a phone or tablet doesn't work too well. Modify your program to use invisible virtual buttons to control the movement of the bat.

Test and save your project.

GetSpriteDistance()

You can determine the distance between the bounding areas of two sprites using the `GetSpriteDistance()` statement (see FIG-17.95).

FIG-17.95

float `GetSpriteDistance` (`(` `id1` `,` `id2` `)`)

`GetSpriteDistance()`

where

id1 is an integer giving the ID of the first sprite.

id2 is an integer giving the ID of the second sprite.

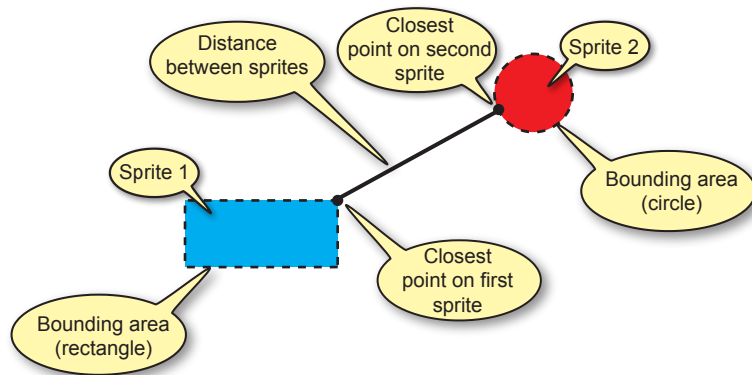
The function returns the shortest distance between the bounding areas of the two sprites.

If the sprites are overlapping, then a value of less than zero will be returned. The distance will be given in the units being used by the app (percentage or virtual pixels).

A typical setup is shown in FIG-17.96.

FIG-17.96

How Distance is
Calculated



After making a call to `GetSpriteDistance()`, the coordinates of the closest points on each of the sprites can be found using the `GetSpriteDistancePoint1X()`, `GetSpriteDistancePoint1Y()`, `GetSpriteDistancePoint2X()`, and `GetSpriteDistancePoint2Y()` statements. The formats of these statements are shown in FIG-17.97.

FIG-17.97

`GetSpriteDistancePoint1X()`
`GetSpriteDistancePoint1Y()`
`GetSpriteDistancePoint2X()`
`GetSpriteDistancePoint2Y()`

```
float GetSpriteDistancePoint1X ( )
float GetSpriteDistancePoint1Y ( )
float GetSpriteDistancePoint2X ( )
float GetSpriteDistancePoint2Y ( )
```

One scenario where the distance between two sprites might be of interest is where one sprite represents a motion sensor. The program in FIG-17.98 causes the first sprite to “light up” when a second sprite is dragged close to it.

FIG-17.98

Using the Distance
Between Sprites

```
rem *** Checking Sprite Distance ***

rem *** Load images ***
LoadImage(1,"Detector.png")
LoadImage(2,"Triangle.png")
rem *** Create Detector ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
SetSpritePositionByOffset(1,50,50)

rem *** Create moveable sprite ***
CreateSprite(2,2)
SetSpriteSize(2,6,-1)
SetSpritePositionByOffset(2,GetPointerX(),GetPointerY())

do
    rem *** Move sprite to mouse position ***
    SetSpritePositionByOffset(2,GetPointerX(),GetPointerY())
    rem *** IF sprites close, show red ***
    if GetSpriteDistance(1,2) < 10
        SetSpriteColorRed(1,255)
    else
        SetSpriteColorRed(1,0)
    endif
    Sync()
loop
```


Activity 17.34

Start a new project called *SpriteDistance* and implement the code given in FIG-17.98 copying *Detector.png* and *Triangle.png* files from *AGKDownloads/Chapter17* to the *media* folder.

Test and save your project.

Controlling Speed

When your app is run on various devices, the speed at which it executes is likely to vary depending on the hardware used. This could be a problem if a game ends up running so fast that the player can't control it! AGK offers various commands to help with this problem.

GetFrameTime()

The `GetFrameTime()` returns the time in seconds between calls to `Sync()`. The statement has the format shown in FIG-17.99.

FIG-17.99

GetFrameTime()

float `GetFrameTime()`

The program in FIG-17.100 is a variation on the earlier bouncing ball project, but this time displaying the time taken to construct each frame.

FIG-17.100

Displaying Frame Time

```
rem *** Frame Time ***

rem *** Create text object ***
CreateText(1,"")

rem *** Load image ***
LoadImage(1,"Ball.png")

rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)

rem *** Set velocity ***
xoffset# = Random(0,40)/10.0 -2
yoffset# = Random(0,40)/10.0 -2

do
    SetSpritePosition(id,GetSpriteX(id)+xoffset#,
        ↵GetSpriteY(id)+yoffset#)
    if GetSpriteX(id) >= (100-GetSpriteWidth(id)) or
        ↵GetSpriteX(id) <= 0
        xoffset# = -xoffset#
    elseif GetSpriteY(id) >= (100-GetSpriteHeight(id)) or
        ↵GetSpriteY(id) <= 0
        yoffset# = -yoffset#
    endif
    SetTextString(1,Str(GetFrameTime(),3))
    Sync()
loop
```

Activity 17.35

Modify your earlier *MovingBall2* project to match the code given in FIG-17.100. Test and save your project.

ScreenFPS()

In a complex program, the time taken between `Sync()` calls may vary considerably depending on the program logic, so finding the time between calls may not give us an accurate picture of the time taken to build the screen display.

An estimate of the number of frames displayed every second (based on the time taken to show the last few frames) can be discovered using the `ScreenFPS()` statement (see FIG-17.101).

FIG-17.101

ScreenFPS()

float `ScreenFPS` ()

Activity 17.36

Modify *MovingBall2* to display the frame rate rather than the time between `Sync()` calls. Test and save your project.

SetSyncRate()

Perhaps the easiest way to control the speed of your game is to explicitly set the frame rate. This can be used to reduce all possible devices to the same speed.

FIG-17.102

SetSyncRate()

To set the number of frames shown per second, use `SetSyncRate()` (see FIG-17.102).

`SetSyncRate` (`ifps` , `iop`)

where

ifps is an integer value giving the display rate in frames per second.

iop is an integer value (0 or 1) used to decide how any delays that need to be introduced to achieve the desired frame rate are to be handled. (0: sleep mode - save CPU and battery; 1: loop until required time has passed - uses CPU but may be more accurate).

The CPU (Central Processing Unit) hardware is responsible for executing all program code.

Activity 17.37

Modify *MovingBall2* setting the frame rate to 50 fps. Try using both the 0 and 1 options for the parameter and observe any differences between the two settings when the program runs. Save your project.

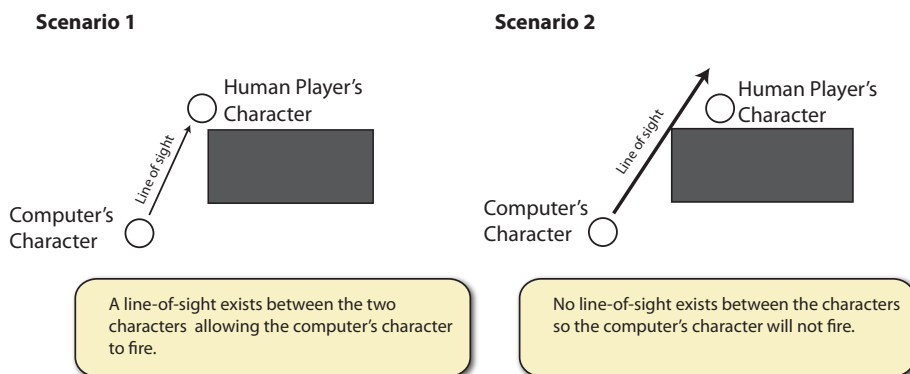
Ray Casting

Determining if there is a line-of-sight between two or more objects can be an important consideration in some games. For example, when you are creating code which determines how a non-human opponent is to react to a player's character,

determining if there is a line-of-sight between the two characters will help decide if the opponent should fire a weapon. This type of situation is shown in FIG-17.103.

FIG-17.103

Using Ray Casting



To help with this type of problem, AGK offers various **ray cast** instructions. You can think of a ray cast as a projected beam of invisible light starting at a specific point (usually from the position of a sprite). If that light hits another object, then that object must be “visible” to anything positioned at the start of the beam.

SpriteRayCast()

To initiate a ray cast between two points, use the `SpriteRayCast()` statement. The statement has the format shown in FIG-17.104.

FIG-17.104

SpriteRayCast()

integer `SpriteRayCast` ((`x1` , `y1` , `x2` , `y2`))

where:

x1,y1 are a pair of real values giving the coordinates of the starting point of the ray cast.

x2,y2 are a pair of real values giving the coordinates of the finishing point of the ray cast.

In fact, the ray cast detects the sprite's bounding area and not the actual sprite itself. The command will not work with the default bounding area assigned to a sprite. This allows sprites to be invisible to the ray cast if you wish. Instead, you must use a bounding area assignment statement such as `SetSpriteShape()` to assign a bounding area to any sprite that you wish to detect using ray casting.

The program in FIG-17.105 sets up three sprites and displays the result of a call to `RayCast()` with the ray starting in the centre of the first object and ending at the top of the app window.

FIG-17.105

Ray Cast Example

```
rem *** Ray casting ***

rem ** Set screen colour ***
SetClearColor(120,120,120)
Sync()

rem *** Create text ***
CreateText(1,"")
SetTextSize(1,3)
```



FIG-17.105

(continued)

Ray Cast Example

```

rem *** Load Images ***
LoadImage(1,"Turret.png")
LoadImage(2,"Tile.png")
LoadImage(3,"Cherry.png")

rem *** Create Sprites ***
rem *** Turret ***
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,42.5,42.5)
rem *** Tile ***
CreateSprite(2,2)
SetSpriteSize(2,10,-1)
SetSpritePosition(2,45,5)
SetSpriteShape(2,2) //Bounding box
rem *** Cherry ***
CreateSprite(3,3)
SetSpriteSize(3,10,-1)
SetSpritePosition(3,45,20)
SetSpriteShape(3,1) //Bounding circle

rem *** Raycast from turret to top of screen ***
hit = SpriteRayCast(GetSpriteXByOffset(1),
  ↳GetSpriteYByOffset(1),50,0)
rem *** Display result of raycast ***
SetTextString(1,Str(hit))

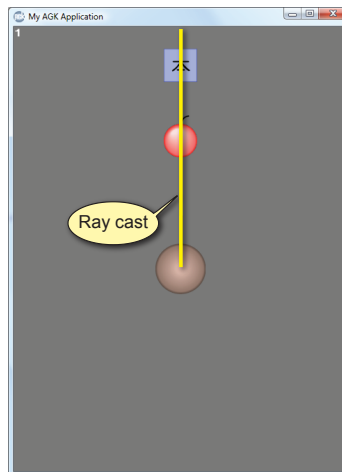
do
  Sync()
loop

```

The setup created by this program is shown in FIG-17.106.

FIG-17.106

The Ray Cast by the Program



Activity 17.38

Start a new project called *RayCasting* and implement the code shown in FIG-17.105.

Run the program. Notice that the text displays a “1” to indicate that the ray has intersected at least one sprite. Save your project.

GetRayCastSpriteID()

If the element intersected by the ray cast is a sprite, then we can find its ID using the `GetRayCastSpriteID()` statement (see FIG-17.107).

FIG-17.107

integer `GetRayCastSpriteId () ()`

`GetRayCastSpriteID()`

The returned value is the ID of the first sprite intersected by the ray cast. If no sprite is encountered, zero is returned.

Activity 17.39

Modify *RayCasting* so that the ID of the intersected sprite is displayed. Test your program. What value is displayed?

Comment out the `SetSpriteShape()` statement for the cherry and see how this changes the result obtained.

Remove the comment characters from the `SetSpriteShape()` statement so that the original result is displayed.

Test and save your project.

GetRayCastX() and GetRayCastY()

You can determine the point at which the ray cast and sprite intersect using the `GetRayCastX()` and `GetRayCastY()` statements (see FIG-17.108).

FIG-17.108

`GetRayCastX()`

float `GetRayCastX () ()`

`GetRayCastY()`

float `GetRayCastY () ()`

Between them, the two functions return the coordinates of the intersection. The coordinates are given in percentage or virtual pixels as determined by the initial app window setup.

Activity 17.40

Modify *RayCasting* so that the coordinates of the intersection are displayed in addition to the sprite ID.

See how this changes when you comment out the `SetSpriteShape()` for the cherry. Remove the comment.

Test and save your project.

We can see the actual position of the ray intersection if we were to place a small circle at the coordinates given.

Activity 17.41

In *Raycasting*, load the image *Spot.png* into a sprite (size 1x1) and place it so that its centre is at the point where the ray intersects with the *cherry* sprite.

Test and save your program.

The point at which the ray cast hits the sprite can be of use if we wanted to create an effect at the point of impact. For example, if we had used the ray cast as the trajectory for a bullet, we could use this intersection point to position an image of a dust cloud.

GetRayCastFraction()

Another piece of information that can be retrieved about a ray cast is how far along the line the intersection occurred. This is given as a fraction of the length of the line specified in `SpriteRayCast()`. This type of information may be useful to determine if the detected sprite is within firing range of a weapon. The statement that returns this information is `GetRayCastFraction()` (see FIG-17.109).

FIG-17.109

float `GetRayCastFraction` ()

`GetRayCastFraction()`

The value returned will lie between 0 (start of line) and 1 (end of line).

Activity 17.42

Change *RayCasting* so that the distance from the start of the ray cast to the point of intersection with the cherry is displayed. Remember the length of the full ray cast is 50. No other data need be displayed.

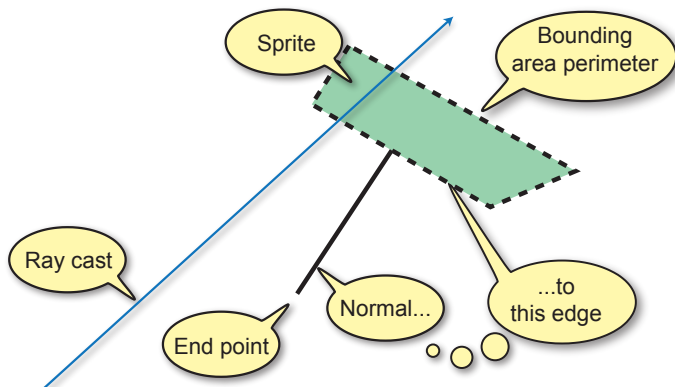
Save your project.

GetRayCastNormalX() and GetRayCastNormalY()

In 2D space we can define a **normal** as a line perpendicular to one edge of a shape. In AGK we can discover details about the normal to the first bounding area edge of any sprite hit by a ray cast. FIG-17.110 shows the concept.

FIG-17.110

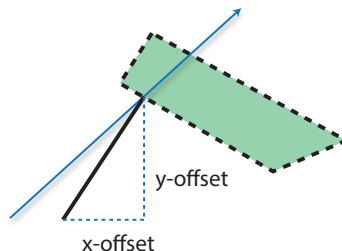
Ray Cast Normal



AGK will give us the offset values from the point at the start of a normal (where the ray cast meets the bounding edge) to the other end of the normal (see FIG-17.111).

FIG-17.111

Ray Cast Offsets



To discover the x and y offsets of the normal, use the statements `GetRayCastNormalX()` and `GetRayCastNormalY()` (see FIG-17.112).

FIG-17.112

`GetRayCastNormalX()`

`GetRayCastNormalY()`

float `GetRayCastNormalX()` ()

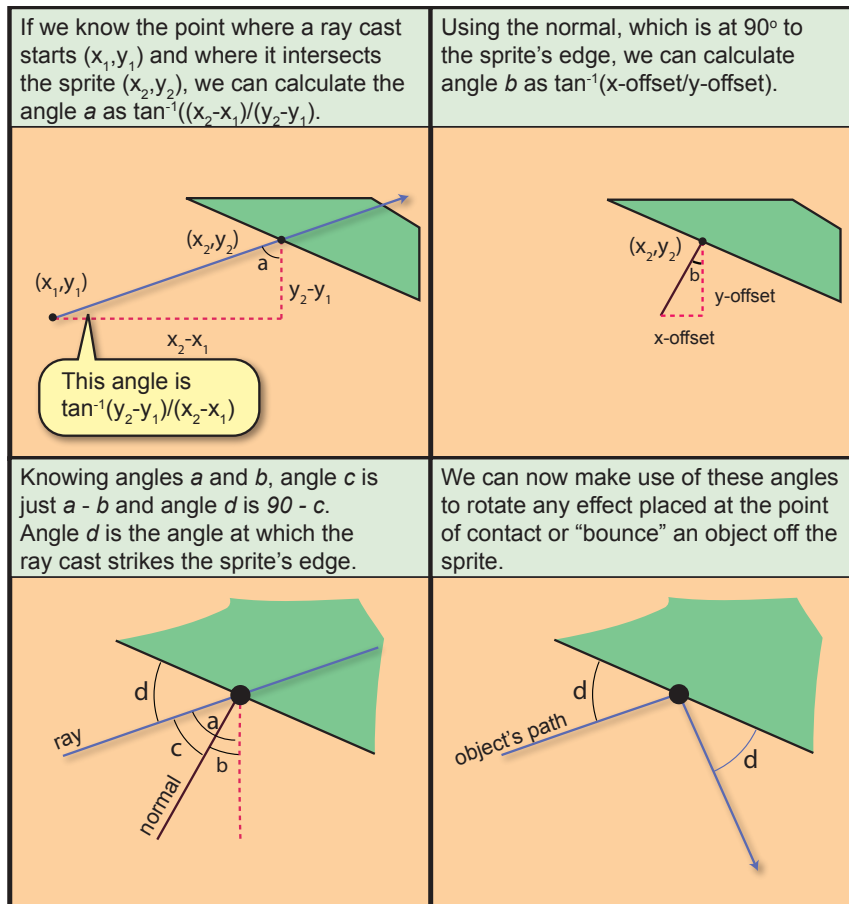
float `GetRayCastNormalY()` ()

By adding the x and y offsets to the intersection point of the ray cast and bounding edge, you can discover the coordinates of the other end of the normal.

So the only question remaining is: what can we do with these details about the normal? The normal allows us to calculate the angle at which an object travelling along the ray cast line would “bounce off” the sprite being hit. This also allows us to adjust the angle of any effect we wish to create (see FIG-17.113).

FIG-17.113

Ray Cast Angles



The program in FIG-17.114 makes use of the angle of the ray cast between the centre of two sprites to rotate a turreted cannon so that it is always pointed at the target sprite. It also makes use of the point at which the ray hits the target to position a cloud of dust symbolising a hit from the cannon.

```

rem *** Ray Cast Angle ***

rem ** Set screen colour ***
SetClearColor(120,120,120)
Sync()

rem *** Load Images ***
LoadImage(1,"Turret2.png") //Firing turret
LoadImage(2,"Shape.png") //Target
LoadImage(3,"Cloud.png") //Hit dust

rem *** Create Sprites ***
rem *** Turret ***
CreateSprite(1,1)
SetSpriteSize(1,10,-1)
SetSpritePosition(1,42.5,42.5)
rem *** Target ***
CreateSprite(2,2)
SetSpriteSize(2,25,-1)
SetSpritePosition(2,40,20)
SetSpriteShape(2,3) //Bounding polygon
rem *** Hit Dust ***
CreateSprite(3,3)
SetSpriteSize(3,5,-1)
SetSpriteOffset(3,GetSpriteWidth(3),GetSpriteHeight(3)/2)

do
    rem *** Move target to pointer ***
    SetSpritePositionByOffset(2,GetPointerX(),GetPointerY())

    rem *** Ray trace turret centre to target centre ***
    hit = SpriteRayCast(GetSpriteXByOffset(1),
        ↵GetSpriteYByOffset(1),GetSpriteXByOffset(2),
        ↵GetSpriteYByOffset(2))

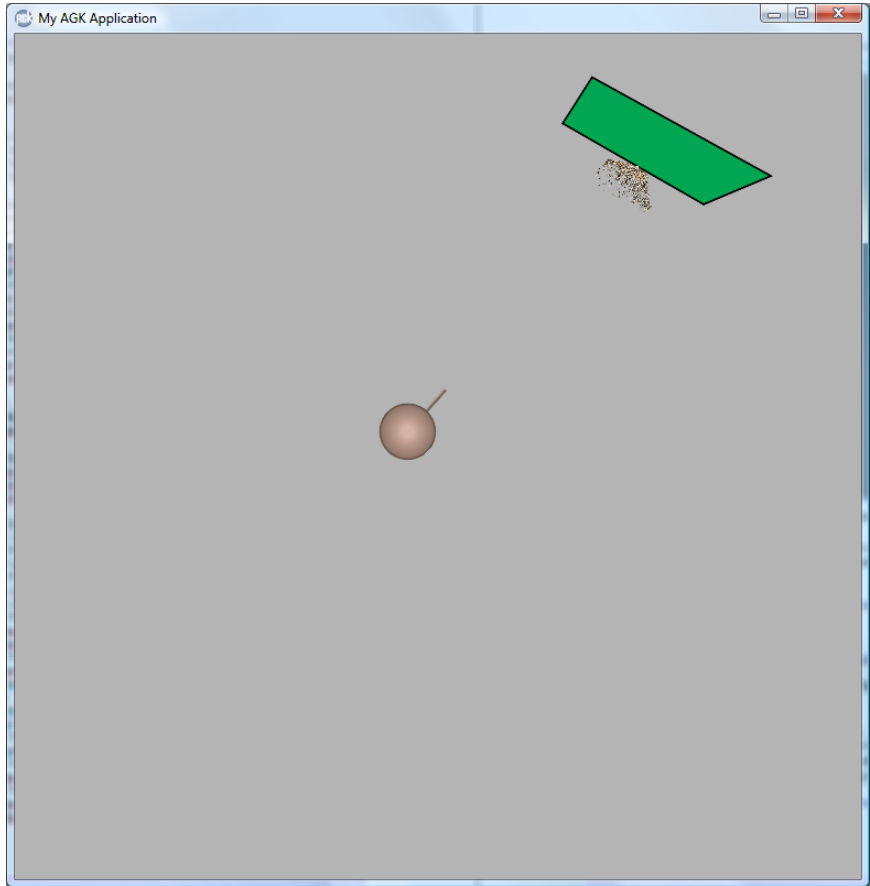
    rem *** If sprite hit (which it must be) ***
    if hit = 1
        rem *** Get ID of sprite hit ***
        id = GetRaycastSpriteID()
        rem *** Get point of collision ***
        x# = GetRayCastX()
        y# = GetRayCastY()
        rem *** Angle turret to point at target ***
        xoffset# = x#-GetSpriteXByOffset(1)
        yoffset# = y#-GetSpriteYByOffset(1)
        rem *** Adjustment for angles outside -90 to +90 ***
        if xoffset# < 0
            bias = 180
        else
            bias = 0
        endif
        rem *** Calculate angle for turret ***
        angle# = ATan(yoffset#/xoffset#)+bias
        rem *** Angle turret to point at target ***
        SetSpriteAngle(1,angle#)
        rem *** Show hit dust (at same angle ***
        SetSpritePositionByOffset(3,x#,y#)
        SetSpriteAngle(3,angle#)
    endif
    Sync()
loop

```


A screenshot of the program is shown in FIG-17.115.

FIG-17.115

Program Screen Dump



There are a few points to note about the program:

The offset of the dust cloud is set to the top-middle. This makes positioning the sprite over the point of collision more accurate. The line required to reposition the offsets is:

```
SetSpriteOffset(3,GetSpriteWidth(3),GetSpriteHeight(3)/2)
```

The line

```
SetSpritePositionByOffset(2,GetPointerX(),GetPointerY())
```

positions the target sprite at the pointer location so it can be dragged to any position on the screen.

The *arctan* function, `ATan()`, only gives results in the range -90° to $+90^\circ$ so we need to adjust this figure when the true angle is outside this range. This happens when the *xoffset#* value is negative, at which time we need to add 180° to the figure returned by `ATan()`. The angle correcting code is

```
rem *** Adjustment for angles outside -90 to +90 ***
if xoffset# < 0
    bias = 180
else
    bias = 0
endif
```

```
rem *** Calculate angle for turret ***
angle# = ATan(yoffset#/xoffset#)+bias
```

Activity 17.43

Start a new project called *CannonFire*, and implement the code given in FIG-17.114. Copy the required files to the project's *media* folder.

Test and save your project.

SpriteRayCastSingle()

FIG-17.116

SpriteRayCastSingle()

If you want to know if a ray cast passes through one particular sprite, then you can use `SpriteRayCastSingle()` which has the format shown in FIG-17.116.

integer `SpriteRayCastSingle` ((`id` , `x1` , `y1` , `x2` , `y2`))

where:

- id** is an integer value giving the ID of the sprite to be checked.
- x1,y1** are a pair of real values giving the coordinates of the starting point of the ray cast.
- x2,y2** are a pair of real values giving the coordinates of the finishing point of the ray cast.

If the ray between the points (*x1*, *y1*), (*x2*, *y2*) passes through sprite *id*, then the statement returns 1, otherwise zero is returned. The sprite need not be the first sprite encountered by the ray.

Summary

- Use `GetSpriteExists()` to check that a sprite of a specified ID exists.
- Use `GetSpriteVisible()` to check on the visibility of a sprite.
- Use `GetSpriteDepth()` to check on the layer on which a sprite is positioned.
- Use `SetSpriteScale()` to resize a sprite in both dimensions. The top-left corner of the sprite remains at a fixed position on the screen.
- Use `SetSpriteAngle()` or `SetSpriteAngleRad()` to rotate a sprite.
- Use `GetSpriteAngle()` or `GetSpriteAngleRad()` to discover the current rotation of a sprite.
- Use `SetSpriteColor()` to set the red, green, blue and transparency of a sprite.
- Use `SetSpriteRed()`, `SetSpriteGreen()`, `SetSpriteBlue()` or `SetSpriteAlpha()` to modify a colour of a sprite or its transparency.
- Use `GetSpriteRed()`, `GetSpriteGreen()`, `GetSpriteBlue()` or `GetSpriteAlpha()` to discover the current value of a sprite's colour component or its transparency.
- Use `GetSpriteHitTest()` to discover if a specified sprite covers a given point.

- Use `GetSpriteHit()` to discover the ID of any sprite covering a given point.
- Use `SetSpriteX()` and `SetSpriteY()` to set the x and y coordinates of a sprite separately.
- Use `GetSpriteX()` and `GetSpriteY()` to discover the x and y coordinates of a sprite.
- Use `GetSpriteWidth()` and `GetSpriteHeight()` to discover the dimensions of a specified sprite.
- Use `GetSpriteImageID()` to discover the ID of the image used by a sprite.
- Use `SetSpriteImage()` to change the image displayed by a sprite.
- Use `SetSpriteTransparency()` to set the transparency of a sprite.
- Use `SetSpriteFlip()` to flip the image shown on a sprite either vertically or horizontally.
- Use `SetSpriteUVScale()` to shrink or enlarge the image displayed by a sprite. The size of the sprite itself is unaffected.
- Use `SetSpriteUVOffset()` to modify the positioning of the image displayed on a sprite. That is, the top-left corner of the image need not be placed at the top-left corner of the sprite.
- Use `SetImageWrapU()` and `SetImageWrapV()` to state how parts of the image which fall outside the area of the sprite are to be handled. This gives the option to wrap the image round onto the opposite edge.
- Use `SetSpriteUVBorder()` to fine-tune how an image is mapped to the edges of a sprite.
- Use `SetSpriteUV()` to gain complete control over how an image is mapped to a sprite.
- Use `ResetSpriteUV()` to reset sprite mapping to normal.
- Use `GetSpritePixelFromX()` and `GetSpritePixelFromY()` to relate a position on a sprite to a position on the original image used by the sprite.
- Use `GetSpriteXFromPixel()` and `GetSpriteYFromPixel()` to relate a position on the original image to a point on the sprite displaying that image.
- Use `SetSpriteOffset()` to modify the point about which a sprite rotates.
- Use `GetSpriteXByOffset()` and `GetSpriteYByOffset()` to determine the screen coordinates of the point about which a sprite rotates.
- Use `SetSpritePositionByOffset()` to use a sprite's point of rotation when positioning the sprite. Using the default settings, that would mean that the centre of a sprite is moved to the specified position rather than its top-left corner.
- Use `SetSpriteScaleByOffset()` to scale a sprite with its offset point remaining fixed on screen.
- Use `SetSpriteShape()` to set the bounding area shape of a sprite.
- Use `SetSpriteShapeBox()` to set the bounding area of a sprite to be a rectangle.

- Use `SetSpriteShapeCircle()` to set the bounding area of a sprite to be a circle.
- Use `SetSpriteShapePolygon()` to set the bounding area of a sprite to be a polygon.
- Grouping sprites allows easier control over sprite hits.
- Use `SetSpriteGroup()` to assign a sprite to a specific group.
- Use `GetSpriteGroup()` to determine to which group a sprite belongs.
- Use `GetSpriteHitGroup()` to detect only sprite hits belonging to a specific group.
- Use `SetSpriteCategoryBits()` to set the categories to which a sprite belongs.
- Use `SetSpriteCategoryBit()` to modify a single bit within a sprite's category setting.
- Use `GetSpriteHitCategory()` to detect sprite hits belonging to specific categories.
- In some applications, when a sprite exits the screen on one edge, it automatically re-enters the screen on the opposite edge.
- Re-entry can be achieved by resetting the appropriate coordinate (*x* or *y*) or by calculating the trajectory of the sprite.
- When the edge of the screen acts as a barrier, moving sprites rebound off the edge and this is achieved by negating the *x* or *y* component of the trajectory as appropriate.
- Use `GetSpriteCollision()` to determine if two specified sprites have collided.
- Use `GetSpriteDistance()` to calculate the distance between two sprites.
- Use `GetFrameTime()` to determine the time between calls to the `Sync()` function.
- Use `ScreenFPS()` to find out the number of frames per second being produced by a program.
- Use `SetSyncRate()` to set the number of frames created every second.
- Use `SpriteRayCast()` to cast a ray between two points.
- Use `GetRayCastSpriteID()` to determine the ID of the first sprite hit by a ray cast.
- Use `GetRayCastX()` and `GetRayCastY()` to find the point at which a ray cast strikes the bounding perimeter of a sprite.
- Use `GetRayCastFraction()` to discover a strike's distance along the ray cast line.
- Use `GetRayCastNormalX()` and `GetRayCastNormalY()` to discover the offsets of the normal to the strike point.
- Use `SetRayCastSingle()` to discover if a ray cast strikes a specific sprite.

A Jigsaw Puzzle Game

Introduction

An atlas texture file format can be put to good use even when the file in question actually contains a single image. In the jigsaw game that follows, we will treat a single image as if it were an atlas texture image, making the accompanying text file split the image into the pieces of our puzzle.

The Game

Again we have a game based on an existing non-computerised pastime. The game consists of simply positioning the randomly placed parts of an image into their correct positions to reconstruct the original image.

This is not designed as a complete and polished game, but merely to show how little code is required to implement the basic idea.

The Data Files

The picture used in the jigsaw (*Cat.png*) is shown in FIG-17.117.

FIG-17.117

Jigsaw Image



The image, which is 700 pixels by 1000 pixels, contains a blue border to help the player position the pieces.

Although the file contains only a single image, it is treated in the program as an atlas texture file and therefore has an accompanying text file (*Cat subimages.txt*) which splits the image into pieces each measuring 100x100 pixels.

A section of the text file is shown in FIG-17.118.

FIG-17.118

The SubImage Text File
(part of)

```
00.png:0:0:100:100
01.png:100:0:100:100
02.png:200:0:100:100
03.png:300:0:100:100
04.png:400:0:100:100
05.png:500:0:100:100
06.png:600:0:100:100
10.png:0:100:100:100
11.png:100:100:100:100
12.png:200:100:100:100
13.png:300:100:100:100
14.png:400:100:100:100
15.png:500:100:100:100
17.png:600:100:100:100
20.png:0:200:100:100
```

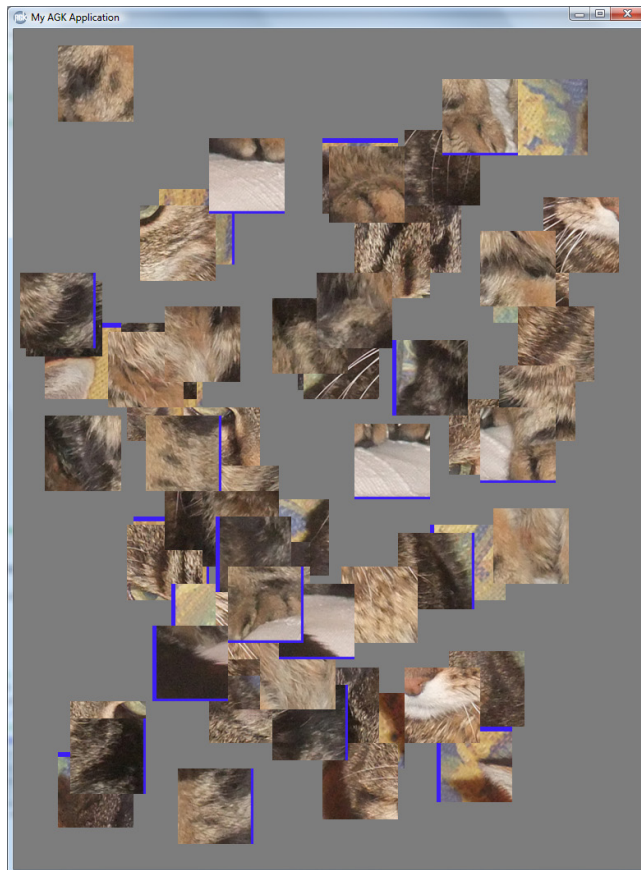
The subimages are named *00.png* through to *96.png* (the second digit of the filename is never greater than 6), so there are 70 sub images in total, giving us 70 pieces in the jigsaw.

Game Layout

The game starts with the pieces scattered randomly about the screen (see FIG-16.119).

FIG-17.119

Jigsaw Start-Up Screen



The player can then drag each piece into position.

Since it is difficult to position a piece exactly, the program snaps each piece to the closest 100x100 position, ensuring an exact fit.

The Game Code

The main program logic calls routines to perform each part of the game:

```
rem *** Main program logic ***
SetUpScreen()
LoadImageUsed()
SplitImageIntoPieces()
CreateSprites()
AllowPlayerToMovePieces()
end
```

As you can see, using appropriately named functions allows us to easily understand the main steps of the program.

This logic is preceded by a few named constants and a single global variable:

```
rem *****
rem ***      Jigsaw Game      ***
rem *****

rem *** Rows and columns in jigsaw ***
#constant NO_OF_ROWS = 10
#constant NO_OF_COLUMNS = 7
rem *** Name of file containing image ***
#constant filename = "Cat.png"
//subimage text file needs to be renamed too
rem *** Global variables ***
rem *** Image ID ***
global id as integer
```

The number of rows and columns are defined as named constants, as is the name of the file containing the image. This allows the values to be easily changed, but you must remember to place any new image and its subimage text file in the *media* folder of the game.

The only global variable (*id*) is the ID assigned to the image. This is needed in more than one routine.

SetUpScreen()

This function sets up the screen's aspect ratio and sets the background colour to grey. It has the following code:

```
function SetUpScreen()
rem *** Set aspect ration ***
SetDisplayAspect(768.0/1024.0)
rem *** Grey background ***
SetClearColor(125,125,125)
endfunction
```

LoadImageUsed()

This function contains only a single line and, as such, it might be argued that it should not be a function at all. The only reason the function has been created is to eliminate any code, other than function calls, from the main program logic. This gives the main logic a neater and more easily understood appearance.

Code for function:

```
function LoadImageUsed()  
    rem *** Load full image ***  
    id = LoadImage(filename)  
endfunction
```

ImageIntoPieces()

This function extracts the subimages from the original picture. The code is:

```
function ImageIntoPieces()  
    rem *** Split image into subimages ***  
    pieceid = 0  
    for row = 0 to NO_OF_ROWS-1  
        for col = 0 to NO_OF_COLUMNS-1  
            inc pieceid  
            LoadSubImage(pieceid,id,str(row)+str(col)+".png")  
        next col  
    next row  
endfunction
```

Notice how the name of each subimage is generated automatically from the phrase

```
str(row)+str(col)+".png"
```

The IDs for the sub images are determined by the variable *pieceid* and will range from 1 to 70 (NO_OF_ROWS x NO_OF_COLUMNS).

CreateSprites()

This function turns each subimage into a sprite and randomly positions it on the screen. The code is:

```
function CreateSprites()  
    rem *** Create sprite for each sub image ***  
    for spriteid = 1 to NO_OF_ROWS * NO_OF_COLUMNS  
        CreateSprite(spriteid,spriteid)  
        SetSpriteSize(spriteid,12,-1)  
        rem *** Set centre of sprite as positioning point ***  
        SetSpriteOffset(spriteid,GetSpriteWidth(spriteid)/2.0,  
            ↵GetSpriteHeight(spriteid)/2.0)  
        SetSpritePosition(spriteid, Random(1,90), Random(1,90))  
    next c  
    Sync()  
endfunction
```

The sprite IDs match those of the corresponding subimage and so lie in the range 1 to 70. Each sprite is made 12% of the screen width. Since there are 7 pieces in each row, a row of pieces will occupy 84% of the screen width, leaving a little space along the edge for pieces that have not yet been placed in position.

AllowPlayerToMovePiece()

The final routine called from the main program logic allows the player to select and drag a piece of the jigsaw. Its code is:

```
function AllowPlayerToMovePieces()  
    rem *** Allow player to move pieces ***
```



```

do
    id = GetPieceSelected()
    if id <> 0
        MovePiece(id)
    endif
    Sync()
loop
endfunction

```

As you can see, this contains a `do loop` structure which means that the routine will never be exited. It's actually up to the player to shut down the app when all the pieces are in place (or when he gives up). Two other functions to do the bulk of the work. These are described below.

GetPieceSelected()

This routine returns the ID of any sprite being pressed/clicked on. If no sprite is currently selected, zero is returned. The function's code is:

```

function GetPieceSelected()
    if GetPointerState()= 1
        spriteid = GetSpriteHit(GetPointerX(),GetPointerY())
    else
        spriteid = 0
    endif
endfunction spriteid

```

MovePiece()

The final function allows the selected subimage to be dragged to a new position. Its code is:

```

function MovePiece(id)
    rem *** Calculate how far the touched part of the ***
    rem *** sprite is from the centre of the sprite ***
    diffx = GetPointerX()-GetSpriteX(id)
    diffy = GetPointerY()-GetSpriteY(id)
    rem *** Bring the sprite to the front layer ***
    SetSpriteDepth(id,0)
    rem *** Allow sprite to be dragged ***
    repeat
        rem *** Take into account how far the touched area
        rem *** is from the centre of the sprite when moving
        rem *** the sprite ***
        SetSpritePosition(id,GetPointerX()-diffx,
            ↵GetPointerY()-diffy)
        Sync()
    until GetPointerState() = 0
    rem *** When the sprite is dropped round off its position
    rem *** to a multiple of the sprite's size ***
    SetSpritePosition(id, Round(GetSpriteX(id)/
        ↵GetSpriteWidth(id))*GetSpriteWidth(id),
        ↵Round(GetSpriteY(id) / GetSpriteHeight(id))*
        ↵GetSpriteHeight(id))
    rem *** Return sprite to original depth ***
    SetSpriteDepth(id,10)
endfunction

```

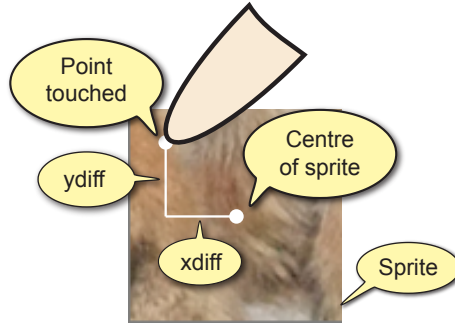
There are several points of interest in this function. Firstly, the distance from the point touched to the centre of the sprite is calculated with the lines:

```
diffx = GetPointerX()-GetSpriteX(id)
diffy = GetPointerY()-GetSpriteY(id)
```

Exactly what is being calculated here is made clearer in FIG-17.120.

FIG-17.120

Touched Point's Offset
from the Sprite Centre



In the `CreateSprites()` function, we added an offset to each sprite so that when a sprite is moved it is always the centre of the sprite that is placed at the specified position. However, if we were to move the sprite directly to the position touched when the player selects a piece of the jigsaw to be moved, then it would be the centre of the sprite that would be placed at this position. This would cause a sudden “jump” of the selected sprite when the screen is first touched. To avoid this jump, we calculate the *xdiff* and *ydiff* values when the screen is first touched and maintain this difference between the touched point and the sprite centre while the image is dragged, thereby creating a much smoother effect.

While the sprite is being dragged, it is repositioned on layer zero. This ensures that it will not disappear behind some other piece while it is being dragged. Once it is dropped, the piece is returned to the default layer 10.

When the sprite is finally dropped, it will shift position slightly so that its centre is positioned at an exact multiple of the sprite's width and height. This ensures that the pieces of the jigsaw fit exactly without requiring an unrealistic effort from the player. The line of code that achieves this effect is:

```
SetSpritePosition(id, Round(GetSpriteX(id) /
    ↳ GetSpriteWidth(id) ) * GetSpriteWidth(id) ,
    ↳ Round(GetSpriteY(id) / GetSpriteHeight(id) ) *
    ↳ GetSpriteHeight(id) )
```

Activity 17.44

Start a new project called *Jigsaw* and implement the code given over the last few pages. Remember to copy the *Cat.png* and *Cat subimages.txt* files into the project's *media* folder.

Test and save your project.

Solutions

Activity 17.1

No solution required.

Activity 17.2

Modified code for *ControllingSprites*:

```
rem *** Controlling a sprite ***
SetClearColor(255,255,255)
rem *** Load sprite image ***
LoadImage(1,"Arrow.png",0)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Size sprite ***
SetSpriteSize(1,7,-1)
rem *** Position sprite ***
SetSpritePosition(1,50,50)
degrees# = 0
do
    degrees# = degrees# +1
    SetSpriteAngle(1,degrees#)
    Sync()
loop
```

Activity 17.3

```
rem *** Controlling a sprite ***
SetClearColor(255,255,255)
rem *** Load sprite image ***
LoadImage(1,"Arrow.png",0)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Size sprite ***
SetSpriteSize(1,7,-1)
rem *** Position sprite ***
SetSpritePosition(1,50,50)
rem *** Change red tint ***
SetSpriteColorRed(1,0)
degrees# = 0
do
    degrees# = degrees# +1
    SetSpriteAngle(1,degrees#)
    Sync()
loop
```

Activity 17.4

No solution required.

Activity 17.5

Modified code for *ShootingGame*:

```
rem *** Button Sprites ***
rem *** Grey background ***
SetClearColor(200,200,200)
rem *** Load sprite images ***
LoadImage(1,"Right.png",0)
LoadImage(2,"Left.png",0)
LoadImage(3,"Fire.png",0)
LoadImage(4,"Arrow.png",0)
rem *** Create sprites ***
CreateSprite(1,1)
CreateSprite(2,2)
CreateSprite(3,3)
CreateSprite(4,4)
rem *** Size sprites ***
SetSpriteSize(1,12,-1)
SetSpriteSize(2,12,-1)
SetSpriteSize(3,12,-1)
SetSpriteSize(4,6,-1)
rem *** Position sprites ***
SetSpritePosition(1,87,92)
SetSpritePosition(2,72,92)
SetSpritePosition(3,1,92)
SetSpritePosition(4,46,80)
red = GetSpriteColorRed(1)
do
    if GetPointerState() = 1
        id = GetSpriteHit(GetPointerX(),
```

```
        GetPointerY())
        if id <> 0
            SetSpriteColorRed(id,200)
            oldid = id
            Sync()
            rem *** If right button, move right ***
            if id = 1
                SetSpriteX(4,GetSpriteX(4)+1)
            endif
            rem *** If left button, move left ***
            if id = 2
                SetSpriteX(4,GetSpriteX(4)-1)
            endif
        endif
    else
        if oldid <> 0
            SetSpriteColorRed(oldid,red)
            oldid = 0
            Sync()
        endif
    endif
    Sync()
loop
```

The arrow moves off the edge of the screen if you continue to press the direction keys

The `do...loop` code within the program is changed as follows:

```
do
    if GetPointerState() = 1
        id = GetSpriteHit(GetPointerX(),GetPointerY())
        if id <> 0
            SetSpriteColorRed(id,200)
            oldid = id
            Sync()
            rem *** IF right button, move right ***
            if id = 1 and GetSpriteX(4) < 94
                SetSpriteX(4,GetSpriteX(4)+1)
            endif
            rem *** IF left button, move left ***
            if id = 2 and GetSpriteX(4) > 0
                SetSpriteX(4,GetSpriteX(4)-1)
            endif
        endif
    else
        if oldid <> 0
            SetSpriteColorRed(oldid,red)
            oldid = 0
            Sync()
        endif
    endif
    Sync()
loop
```

Activity 17.6

Modified code for *SwapImage*:

```
rem *** Swap sprite image ***
rem *** Load Images ***
LoadImage(1,"Round.png")
LoadImage(2,"Square.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
imageID = 1
do
    rem *** If pointer pressed and over sprite ***
    if GetSpriteHit(GetPointerX(),GetPointerY()) and
        GetPointerPressed()
        rem *** Change to other image ***
        imageID = 3 - imageID
        SetSpriteImage(1,imageID)
    endif
    Sync()
loop
```

Activity 17.7

The only change required in the program code is that the second `LoadImage()` statement is changed to:

```
LoadImage(2,"FourCircles.png")
```

The four circles image is distorted making the circles appear as ellipses.

Activity 17.8

Modified code for *SwapImage*:

```

rem *** Swap sprite image ***

rem *** Load Images ***
LoadImage(1,"Round.png")
LoadImage(2,"FourCircles.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
imageID = 1
do
    rem *** If pointer pressed and over sprite ***
    if GetSpriteHit(GetPointerX(),GetPointerY()) and
        ⌘GetPointerPressed()
        rem *** Change to other image ***
        imageID = 3 - imageID
        SetSpriteImage(1,imageID)
        SetSpriteSize(1,12,-1)
    endif
    Sync()
loop

```

The image now displays the correct ratio (the circles appear as circular) but has less height than the first image.

Activity 17.9

No solution required.

Activity 17.10

No solution required.

Activity 17.11

No solution required.

Activity 17.12

Code for *UOffset*:

```

rem *** Offset values ***
dim offsets[12] as float = [0,0,0,0.25,0,0.5,-0.5,
⌘0.7,0.25,-0.25,0,-0.5,-0.25]
rem *** Grey screen ***
SetClearColor(120,120,120)
Sync()
rem *** Text resource ***
CreateText(1,"")
rem *** Load image and create sprite ***
LoadImage(1,"DS.png")
CreateSprite(1,1)
SetSpriteSize(1,30,-1)
SetSpritePosition(1,35,35)
for c = 1 to 12 step 2
    rem *** Set string ***
    SetTextString(1, "Offsets U:"+Str(offsets[c])+
        ⌘" V:"+Str(offsets[c+1]))
    rem *** Change offset ***
    SetSpriteUOffset(1,offsets[c],offsets[c+1])
    Sync()
    Sleep(5000)
next c
do
loop

```

Activity 17.13

Modified code for *UOffset*:

```

rem *** Offset values ***
dim offsets[12] as float = [0,0,0,0.25,0,0.5,0.5,
⌘0.7,0.25,-0.25,0,-0.5,-0.25]
rem *** Grey screen ***
SetClearColor(120,120,120)
Sync()
rem *** Text resource ***
CreateText(1,"")
rem *** Load image and create sprite ***
LoadImage(1,"DS.png")
rem *** Set U and V wrap ***
SetImageWrapU(1,1)

```

```

SetImageWrapV(1,1)
CreateSprite(1,1)
SetSpriteSize(1,30,-1)
SetSpritePosition(1,35,35)
for c = 1 to 12 step 2
    rem *** Set string ***
    SetTextString(1, "Offsets U:"+Str(offsets[c])+
        ⌘" V:"+Str(offsets[c+1]))
    rem *** Change offset ***
    SetSpriteUOffset(1,offsets[c],offsets[c+1])
    Sync()
    Sleep(5000)
next c
do
loop

```

Activity 17.14

The program produces a sprite with the image repeated several times in the horizontal and vertical directions. Those images scroll diagonally from bottom-right to top-left.

Activity 17.15

You should see a subtle shift of the image on the sprite after 5 seconds.

Activity 17.16

No solution required.

Activity 17.17

No solution required.

Activity 17.18

To have the sprite rotate about its bottom-right corner, the line

```
SetSpriteOffset(2,0,0)
```

must be changed to

```

SetSpriteOffset(2,GetSpriteWidth(2),
⌘GetSpriteHeight(2))

```

Activity 17.19

To have the sprite rotate about a point above and to its left, the line

```

SetSpriteOffset(2,GetSpriteWidth(2),
⌘GetSpriteHeight(2))

```

must be changed to

```
SetSpriteOffset(2,-15,-15)
```

Activity 17.20

Modified code for *SpriteOffset*:

```

rem *** Sprite Offsets ***
rem *** Create text resource ***
CreateText(1,"")
SetTextSize(1,3)
rem *** Grey background ***
SetClearColor(100,100,100)
Sync()
rem *** Background grid image ***
LoadImage(1,"Grid.png")
rem *** Test Image ***
LoadImage(2,"Arrows.png")
rem *** Create grid in background ***
CreateSprite(1,1)
SetSpriteSize(1,100,100)
rem *** Add arrows sprite ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,40,40)
Sync()
Sleep(1000)

```

```

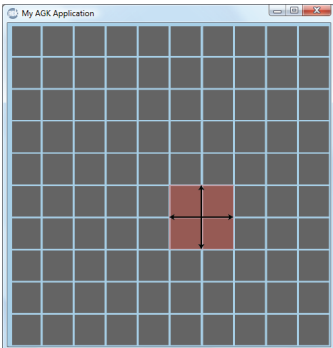
rem *** Display offset location ***
SetTextString(1,"Rotation offset X: "
↳+Str(GetSpriteXByOffset(2))+" Y: "
↳+Str(GetSpriteYByOffset(2)))
rem *** Rotate sprite about its centre ***
for c = 0 to 360
    SetSpriteAngle(2,c)
    Sync()
next c
Sleep(1000)
Sync()
rem *** Move rotation point to
rem *** above and left of sprite ***
SetSpriteOffset(2,-15,-15)
SetSpritePosition(2,40,40)
rem *** Display offset location ***
SetTextString(1,"Rotation offset X: "
↳+Str(GetSpriteXByOffset(2))+" Y: "
↳+Str(GetSpriteYByOffset(2)))
Sync()
Sleep(1000)
rem *** Rotate sprite about new point ***
for c = 0 to 360
    SetSpriteAngle(2,c)
    Sync()
next c
do
loop

```

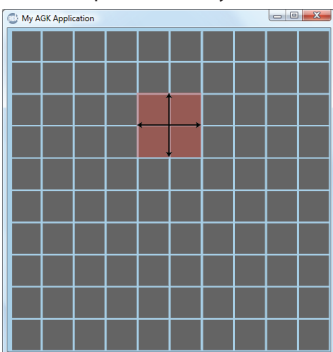
Initially, the sprite rotates about its own centre giving the result (40,40). In the second part of the program, the sprite's offset is moved 15 units up and 15 units to the left (measured from the top-left of the sprite). This position is outside the area of the sprite. It is this point (25,25) which the sprite then rotates about.

Activity 17.21

Original Position



After SetSpritePositionByOffset()



Centre of sprite at (50,30)

Using `SetSpritePosition()` places the sprite's top-left corner at position (50,50); using `SetSpritePositionByOffset()` places the sprite's centre at (50,50).

Activity 17.22

No solution required.

Activity 17.23

The vertex marker sprite is always at the pointer's current position. This is done with the line:

```

SetSpritePositionByOffset(3,GetPointerX(),
↳GetPointerY())

```

When the mouse button is clicked, a new copy of the sprite is created and the previous sprite left at its current position. This is done by the line:

```

CloneSprite(nextsprite,3)

```

Each vertex's coordinates are stored in the `vertices` array.

The bounding polygon is set up with the lines:

```

vcount = count / 2
for c = 0 to vcount-1
    SetSpriteShapePolygon(1,vcount, c,vertices[c*2],
↳vertices[c*2+1])
next c

```

Activity 17.24

No solution required.

Activity 17.25

To allow off-board pieces to be selected at any time change the following lines:

Old: `activecategory = 2 //On-board red`

New: `activecategory = 10 //Red and off-board`

Old: `activecategory = 6 - activecategory`

New: `activecategory = 22 - activecategory`

Activity 17.26

Modified code for *MovingBall* (move to left):

```

rem *** Moving a sprite ***

rem *** Load image ***
LoadImage(1,"Ball.png")
rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)
rem *** Set Speed ***
speed = -1
rem *** Move sprite ***
do
    SetSpritePosition(id,GetSpriteX(id)+speed,
↳GetSpriteY(id))
    Sync()
loop

```

Modified code for *MovingBall* (move down):

```

rem *** Moving a sprite ***

rem *** Load image ***
LoadImage(1,"Ball.png")
rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)
rem *** Set Speed ***
speed = 2
rem *** Move sprite ***
do
    SetSpritePosition(id,GetSpriteX(id),
↳GetSpriteY(id)+speed)
    Sync()
loop

```

Activity 17.27

Modified code for *MovingBall* (speed 2 angle 60°):

```

rem *** Moving a sprite ***
rem *** Load image ***
LoadImage(1,"Ball.png")
rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)
rem *** Set velocity ***
speed = 2
xoffset# = speed * cos(60)
yoffset# = speed * sin(60)
rem *** Move sprite ***
do
    SetSpritePosition(id,GetSpriteX(id)+xoffset#,
        ↳GetSpriteY(id)+yoffset#)
    Sync()
loop

```

Activity 17.28

Modified code for *MovingBall*:

```

rem *** Moving a sprite ***
rem *** Load image ***
LoadImage(1,"Ball.png")
rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)
rem *** Set velocity ***
speed = 2
xoffset# = Random(0,40)/10.0 -2
yoffset# = Random(0,40)/10.0 -2
rem *** Move sprite ***
do
    SetSpritePosition(id,GetSpriteX(id)+xoffset#,
        ↳GetSpriteY(id)+yoffset#)
    Sync()
loop

```

Activity 17.29

Modified code for *MovingBall*:

```

rem *** Moving a sprite ***
rem *** Load image ***
LoadImage(1,"Ball.png")
rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)
rem *** Set velocity ***
speed = 2
xoffset# = Random(0,40)/10.0 -2
yoffset# = Random(0,40)/10.0 -2
rem *** Move sprite ***
do
    if GetSpriteX(id) >= 100
        SetSpritePosition(id,0,GetSpriteY(id))
    elseif GetSpriteX(id) <= -5
        SetSpritePosition(id,100,GetSpriteY(id))
    elseif GetSpriteY(id) >= 100
        SetSpritePosition(id,GetSpriteX(id),0)
    elseif GetSpriteY(id) <= -GetSpriteHeight(id)
        SetSpritePosition(id,GetSpriteX(id),100)
    endif
    Sync()
loop

```

Activity 17.30

No solution required.

Activity 17.31

No solution required.

Activity 17.32

No solution required.

Activity 17.33

Modified code for *BatAndBall*:

```

rem *** Bat and ball ***

rem *** Load images ***
LoadImage(1,"Ball.png")
LoadImage(2,"Bat.png")

rem *** Create sprites ***
CreateSprite(1,1)
SetSpriteSize(1,4,-1)
SetSpritePosition(1,48,5)
CreateSprite(2,2)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,47.5,95)

rem *** Create joystick ***
SetJoystickScreenPosition(90,40,10)

Sync()
Sleep(2000)

rem *** Set ball's velocity***
yoffset# = Random(3,8)/10.0
xoffset# = (Random(0,40)-20)/10.0

rem *** Set game state to playing (1) ***
gamestate = 1

repeat
    rem *** Redraw ball's position ***
    SetSpritePosition(1,GetSpriteX(1)+xoffset#,
        ↳GetSpriteY(1)+yoffset#)
    rem *** If the sprite hits the left or right
    ↳sides change xoffset ***
    if GetSpriteX(1)<=0 or GetSpriteX(1)>= 100-
        ↳GetSpriteWidth(1)
        xoffset# = -xoffset#
    rem *** If ball hits top or bat, change yoffset
    ↳***
    elseif GetSpriteY(1)<=0 or
        ↳GetSpriteCollision(1,2) = 1
        yoffset# = -yoffset#
    rem *** If sprite passes bottom edge, end game
    ↳***
    elseif GetSpriteY(1) > 100
        DeleteSprite(1)

        gamestate = 0
    endif
    rem *** Move bat ***
    SetSpritePosition(2,GetSpriteX(2)+GetJoystickX()
        ↳,GetSpriteY(2))
    Sync()
until gamestate = 0

rem *** Don't let game terminate ***
do
loop

```

To use virtual buttons, the joystick code is removed from the program and virtual buttons added to the left and right edges by the bat.

Modified code for *BatAndBall*:

```

rem *** Bat and ball ***

rem *** Load images ***
LoadImage(1,"Ball.png")
LoadImage(2,"Bat.png")
rem *** Create sprites ***
CreateSprite(1,1)
SetSpriteSize(1,4,-1)
SetSpritePosition(1,48,5)
CreateSprite(2,2)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,47.5,95)

rem *** Record bat's x-coordinate ***
batx = 47.5
rem *** Create virtual buttons ***
AddVirtualButton(1,5,94,10)
AddVirtualButton(2,95,94,10)
SetVirtualButtonVisible(1,0)
SetVirtualButtonVisible(2,0)

Sync()

```

```

Sleep(2000)
rem *** Set ball's velocity***
yoffset# = Random(3,8)/10.0
xoffset# = (Random(0,40)-20)/10.0
rem *** Set game state to playing (1) ***
gamestate = 1
repeat
    rem *** Redraw ball's position ***
    SetSpritePosition(1,GetSpriteX(1)+xoffset#,
        ↵GetSpriteY(1)+yoffset#)
    rem *** If the sprite hits the left or right
    ↵sides change xoffset ***
    if GetSpriteX(1)<=0 or GetSpriteX(1)>= 100-
        ↵GetSpriteWidth(1)
        xoffset# = -xoffset#
    rem *** If ball hits top or bat, change yoffset
    ↵***
    elseif GetSpriteY(1)<=0 or
        ↵GetSpriteCollision(1,2) = 1
        yoffset# = -yoffset#
    rem *** If sprite passes bottom edge, end game
    ↵***
    elseif GetSpriteY(1) > 100
        DeleteSprite(1)
        gamestate = 0
    endif
    rem *** Move bat ***
    if GetVirtualButtonState(1) = 1
        dec batx
    elseif GetVirtualButtonState(2) = 1
        inc batx
    endif
    SetSpritePosition(2,batx,GetSpriteY(2))
    Sync()
until gamestate = 0
rem *** Don't let game terminate ***
do
loop

```

Activity 17.34

No solution required.

Activity 17.35

No solution required.

Activity 17.36

Modified code for *MovingBall2*:

```

rem *** Frame Rate ***
rem *** Create text object ***
CreateText(1,"")
rem *** Load image ***
LoadImage(1,"ball.png")
rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)
rem *** Set velocity ***
xoffset# = Random(0,40)/10.0 -2
yoffset# = Random(0,40)/10.0 -2
do
    SetSpritePosition(id,GetSpriteX(id)+xoffset#,
        ↵GetSpriteY(id)+yoffset#)
    if GetSpriteX(id) >= (100-GetSpriteWidth(id)) or
        ↵GetSpriteX(id) <= 0
        xoffset# = -xoffset#
    elseif GetSpriteY(id) >= (100-
        ↵GetSpriteHeight(id)) or GetSpriteY(id) <= 0
        yoffset# = -yoffset#
    endif
    rem *** Display frame rate ***
    SetTextString(1,Str(ScreenFPS(),3))
    Sync()
loop

```

Activity 17.37

Modified code for *MovingBall2*:

```

rem *** Frame Rate ***
rem *** Set Sync rate ***
SetSyncRate(50,0)

```

```

rem *** Create text object ***
CreateText(1,"")
rem *** Load image ***
LoadImage(1,"ball.png")
rem *** Create, size and position sprite ***
id = CreateSprite(1)
SetSpriteSize(id,5,-1)
SetSpritePosition(id,47.5,47.5)
rem *** Set velocity ***
xoffset# = Random(0,40)/10.0 -2
yoffset# = Random(0,40)/10.0 -2
do
    SetSpritePosition(id,GetSpriteX(id)+xoffset#,
        ↵GetSpriteY(id)+yoffset#)
    if GetSpriteX(id) >= (100-GetSpriteWidth(id)) or
        ↵GetSpriteX(id) <= 0
        xoffset# = -xoffset#
    elseif GetSpriteY(id) >= (100-
        ↵GetSpriteHeight(id)) or GetSpriteY(id) <= 0
        yoffset# = -yoffset#
    endif
    rem *** Display frame rate ***
    SetTextString(1,Str(ScreenFPS(),3))
    Sync()
loop

```

Changing to `SetSyncRate(50,1)` seems to give a slightly smoother effect.

Activity 17.38

No solution required.

Activity 17.39

To display the sprite's ID change the end of the program to read:

```

rem *** Raycast from turret to top of screen ***
hit = SpriteRayCast(GetSpriteXByOffset(1),
    ↵GetSpriteYByOffset(1),50,0)
if hit = 1
    id = GetRaycastSpriteID()
    rem *** Display ID of sprite hit ***
    SetTextString(1,Str(id))
endif
do
    Sync()
loop

```

The program displays 3, the ID of the cherry.

When the new bounding area is removed from the cherry sprite, it is ignored by the ray cast and the tile's ID (2) is displayed.

Activity 17.40

To display the coordinates at which the ray strikes the cherry sprite, change the end of the program to read:

```

rem *** Raycast from turret to top of screen ***
hit = SpriteRayCast(GetSpriteXByOffset(1),
    ↵GetSpriteYByOffset(1),50,0)
if hit = 1
    id = GetRaycastSpriteID()
    x# = GetRaycastX()
    y# = GetRaycastY()
    rem *** Display ID of sprite hit and coords ***
    SetTextString(1,Str(id)+" Hit at position ("
        ↵+Str(x#,1)+" "+Str(y#,1)+"")
endif
do
    Sync()
loop

```

Activity 17.41

Modified code for *Raycasting*:

```

rem *** Ray casting ***

rem ** Set screen colour ***
SetClearColor(120,120,120)

```

```

Sync()

rem *** Create text ***
CreateText(1,"")
SetTextSize(1,3)

rem *** Load Images ***
LoadImage(1,"Turret.png")
LoadImage(2,"Tile.png")
LoadImage(3,"Cherry.png")
rem *** Spot marker ***
LoadImage(4,"Spot.png")

rem *** Create Sprites ***
rem *** Turret ***
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,42.5,42.5)
rem *** Tile ***
CreateSprite(2,2)
SetSpriteSize(2,10,-1)
SetSpritePosition(2,45,5)
SetSpriteShape(2,2) //Bounding box
rem *** Cherry ***
CreateSprite(3,3)
SetSpriteSize(3,10,-1)
SetSpritePosition(3,45,20)
SetSpriteShape(3,1) //Bounding circle
rem *** Raycast from turret to top of screen ***
hit = SpriteRayCast(GetSpriteXByOffset(1),
↳GetSpriteYByOffset(1),50,0)
if hit = 1
    id = GetRaycastSpriteID()
    x# = GetRaycastX()
    y# = GetRaycastY()
    rem *** Mark spot hit ***
    CreateSprite(4,4)
    SetSpriteSize(4,1,-1)
    SetSpritePositionByOffset(4,x#,y#)
    rem *** Display ID of sprite hit and coords ***
    SetTextString(1,Str(id)+" Hit at position ("
↳Str(x#,1)+"", "+Str(y#,1)+"")
endif
do
    Sync()
loop

```

Activity 17.42

To display the distance from the start of the ray to the first sprite hit, change the end of the *Raycasting* program to:

```

rem *** Raycast from turret to top of screen ***

hit = SpriteRayCast(GetSpriteXByOffset(1),
↳GetSpriteYByOffset(1),50,0)
if hit = 1
    id = GetRaycastSpriteID()
    rem *** Calculate distance ***
    distance# = GetRayCastFraction()*50 //Fraction
    ↳ length of ray
    rem *** Mark spot hit ***
    CreateSprite(4,4)
    SetSpriteSize(4,1,-1)
    SetSpritePositionByOffset(4,x#,y#)
    rem *** Display distance to sprite from start of
    ↳ ray cast ***
    SetTextString(1,"Distance to sprite "
↳Str(distance#,1))
endif
do
    Sync()
loop

```

Activity 17.43

No solution required.

Activity 17.44

Code for *Jigsaw*:

```

rem *****
rem *** Jigsaw Game ***
rem *****

rem *** Named constants ***

```

```

rem *** Rows and columns in jigsaw ***
#constant NO_OF_ROWS = 10
#constant NO_OF_COLUMNS = 7

rem *** Name of file containing image ***
#constant filename = "Cat.png"
//subimage text file needs to be renamed too

rem *** Global variables ***
rem *** Image ID ***
global id as integer

```

```

rem *** Main program logic ***
SetUpScreen()
LoadImageUsed()
SplitImageIntoPieces()
CreateSprites()
AllowPlayerToMovePieces()
end

```

```

rem *** Level 1 Functions ***

```

```

function SetUpScreen()
    rem *** Set aspect ration ***
    SetDisplayAspect(768.0/1024.0)
    rem *** Grey background ***
    SetClearColor(125,125,125)
endfunction

```

```

function LoadImageUsed()
    rem *** Load full image ***
    id = LoadImage(filename)
endfunction

```

```

function SplitImageIntoPieces()
    rem *** Split image into subimages ***
    pieceid = 0
    for row = 0 to NO_OF_ROWS-1
        for col = 0 to NO_OF_COLUMNS-1
            inc pieceid
            LoadSubImage(pieceid,id,str(row)+
↳str(col)+".png")
        next col
    next row
endfunction

```

```

function CreateSprites()
    rem *** Create sprite for each sub image ***
    for spriteid = 1 to NO_OF_ROWS * NO_OF_COLUMNS
        CreateSprite(spriteid,spriteid)
        SetSpriteSize(spriteid,12,-1)
        rem *** Set centre of sprite as positioning
        ↳ point ***
        SetSpriteOffset(spriteid,GetSpriteWidth
↳(spriteid)/2.0, GetSpriteHeight(spriteid)
↳/2.0)
        SetSpritePosition(spriteid, Random(1,90),
↳Random(1,90))
    next c
    Sync()
endfunction

```

```

function AllowPlayerToMovePieces()
    rem *** Allow player to move pieces ***
    do
        id = GetPieceSelected()
        if id <> 0
            MovePiece(id)
        endif
        Sync()
    loop
endfunction

```

```

rem *** Level 2 Functions ***

```

```

function GetPieceSelected()
    if GetPointerState() = 1
        spriteid = GetSpriteHit(GetPointerX(),
↳GetPointerY())
    else
        spriteid = 0
    endif
endfunction spriteid

```

```

function MovePiece(id)
    rem *** Calculate how far the touched part of
    ↳ the sprite is from ***

```



```

rem *** the centre of the sprite ***
diffx = GetPointerX()-GetSpriteX(id)
diffy = GetPointerY()-GetSpriteY(id)
rem *** Bring the sprite to the front layer ***
SetSpriteDepth(id,0)
rem *** Allow sprite to be dragged ***
repeat
    rem *** Take into account how far the
    ↳touched area is from the centre ***
    rem *** of the sprite when moving the sprite
    ↳***
    SetSpritePosition(id,GetPointerX()-
    ↳diffx,GetPointerY()-diffy)
    Sync()
until GetPointerState() = 0
rem *** When the sprite is dropped round off its
↳position to a multiple of the sprite's size
↳***
SetSpritePosition(id, Round(GetSpriteX(id)/
↳GetSpriteWidth(id)) *GetSpriteWidth(id),
↳Round(GetSpriteY(id) / GetSpriteHeight(id))*
↳GetSpriteHeight(id))
rem *** Return sprite to original depth ***
SetSpriteDepth(id,10)
endfunction

```


Animated Sprites

In this Chapter:

- ☐ Using Animated Sprites to Create Movement
- ☐ Using Animated Sprites for Alternative Images
- ☐ Adding New Frames to an Animation
- ☐ Playing an Animation
- ☐ Showing a Single Frame of an Animation
- ☐ An Asteroids Game Using Animated Sprites

Animated Sprites

Introduction

An animated sprite is one that consists of more than a single image. Each separate image that goes to make up this collection of pictures is known as a **frame**.

Frames are numbered. The first frame is frame 1.

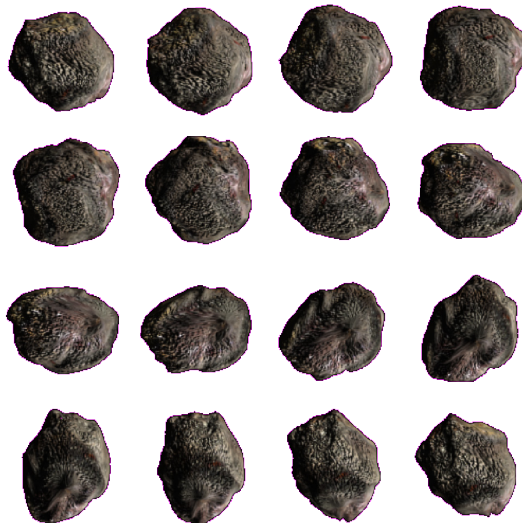
The collection of images in an animation may indeed make up, as the name suggests, an animated display - just like a cartoon or computer generated animation you might watch on television - but the images may be separate, unrelated pictures. For example, we might use two images to represent the two sides of a coin, or six images to represent the six possible throws on a dice; we could even have 52 images to represent a pack of playing cards.

Using an Animated Sprite

Like any sprite, we need to start by obtaining the image we intend use on the sprite. In this first example, we'll make use of a set of images representing a spinning asteroid (image courtesy of The Game Creators) as shown in FIG-18.1.

FIG-18.1

Asteroid Frames



The single image containing the frames of the animation is loaded in the usual way

```
LoadImage(1,"Asteroid.png")
```

and assigned to a sprite:

```
CreateSprite(1,1)  
SetSpriteSize(1,12,-1)  
SetSpritePosition(1,46,46)
```

SetSpriteAnimation()

To convert a normal sprite into an animated one, we need to execute the `SetSpriteAnimation()` statement. This specifies the dimensions of each frame within the animation as well as the total number of frames. FIG-18.2 shows the

format for the `SetSpriteAnimation()` statement.

FIG-18.2

`SetSpriteAnimation()`

`SetSpriteAnimation` (`id` , `iwidth` , `iheight` , `icount`)

where:

- id** is an integer value giving the ID of the sprite.
- iwidth** is an integer value giving the width of a single frame in pixels.
- iheight** is an integer value giving the height of a single frame in pixels.
- icount** is an integer value giving the number of frames in the animation.

Notice that each frame of the animation is assumed to be the same size; this is a restriction you must adhere to.

The asteroid image is 1000 by 1000 pixels and contains 16 frames, so the statement needed for this is:

```
SetSpriteAnimation(1,250,250,16)
```

PlaySprite()

When we have created a sprite designed to give the impression of movement (as is the case with the asteroid sprite), then we can play that animation using the `PlaySprite()` statement. This statement has many optional parameters (see FIG-18.3).

FIG-18.3

`PlaySprite()`

`PlaySprite` (`id` [, `ifps` [, `iloop` [, `istart` , `iend`]]])

where:

- id** is an integer value giving the ID of the animated sprite.
- ifps** is an integer value giving the frames per second play rate. If this is too high for the hardware being used, some frames may be skipped. The default value is 10.
- iloop** is an integer value (0 or 1) which determines if the animation is to be played continuously (1) or only once (0). The default value is 1.
- istart** is an integer value giving the first frame to be played. The default value is 1 (the first frame).
- iend** is an integer value giving the last frame to be played. The default value is the frame number of the last frame.

So, we could play our animation using a line as simple as:

```
PlaySprite(1)
```

However, if we wanted to play the animation only once at a rate of 15 frames per second and using only frame 3 to 10, then we would write:

```
PlaySprite(1,15,0,3,10)
```

Activity 18.1

Start a new project called *Comet*, copy the file *Asteroid.png* from *AGKDownloads/Chapter18* into the project's *media* folder then code the program to display the animated asteroid in the middle of the app window.

Test your project and then modify it so that the animation plays at 20 frames per second.

Save your project.

AddSpriteAnimationFrame()

If you have created your own graphics for an animation, your image will, of course, include all the frames you need, but sometimes, particularly if you are making use of other people's work (as is the case with the asteroid), then you may want to add one or more frames that were not in the original picture. One approach is to modify that original image and add in your new frames. Alternatively, you can add a new frame to the animation from a separate image by using the `AddSpriteAnimationFrame()` statement (see FIG-18.4).

FIG-18.4

AddSpriteAnimation
↳ Frame()

`AddSpriteAnimationFrame (id , imgId)`

where:

id is an integer value giving the ID of the animated sprite.

imgId is an integer value giving the ID of the image to be added to the animation.

The image should be the same size as a single frame of the animation. However, images of other sizes will be scaled to fit the frame size.

Activity 18.2

Copy the three files named *Explode01.png*, *Explode02.png* and *Explode03.png* from rom *AGKDownloads/Chapter18* into project *Comet's media* folder.

Modify *Asteroid* so that these three files are loaded as images and then added as new frames to the asteroid sprite.

Test your program. *The last three frames should only appear when the asteroid is destroyed.*

Modify the animation so that only frames 1 to 16 are shown when the animation plays. Test your code.

Modify your project again so that the three explosion frames play only once after 5 seconds have elapsed.

Test and save your project.

SetSpriteSpeed()

Although a sprite's speed is set when the `PlaySprite()` statement is executed, that speed can be modified as the sprite is playing using the `SetSpriteSpeed()` statement (see FIG-18.5).

FIG-18.5

SetSpriteSpeed()

`SetSpriteSpeed (id , ifps)`

where:

- id** is an integer value giving the ID of the animated sprite.
- ifps** is an integer value giving the frames per second play rate. If this is too high for the hardware being used, some frames may be skipped. A value of zero will cause the animation to pause at the frame currently being displayed.

StopSprite()

A running animation can be stopped (without changing the frame rate) using the `StopSprite()` statement (see FIG-18.6).

FIG-18.6

StopSprite()

`StopSprite (id)`

where:

- id** is an integer value giving the ID of the animated sprite.

ResumeSprite()

A stopped sprite can be restarted using the `ResumeSprite()` statement (see FIG-18.7).

FIG-18.7

ResumeSprite()

`ResumeSprite (id)`

where:

- id** is an integer value giving the ID of the animated sprite.

The sprite will begin playing at the same frame at which it stopped and at the speed previously set.

GetSpritePlaying()

We can check if an animated sprite is currently playing using the `GetSpritePlaying()` statement (see FIG-18.8)

FIG-18.8

GetSpritePlaying()

integer `GetSpritePlaying (id)`

where:

- id** is an integer value giving the ID of the animated sprite.

The statement returns 1 if the sprite is playing, otherwise zero is returned.

GetSpriteFrameCount()

To find out exactly how many frames are in an animation, you can use the `GetSpriteFrameCount()` statement (see FIG-18.9).

FIG-18.9

`GetSpriteFrameCount()`

integer `GetSpriteFrameCount` (`id`)

where:

id is an integer value giving the ID of the animated sprite.

ClearSpriteAnimationFrames()

As we have already seen, the `SetSpriteAnimation()` statement splits the single image assigned to a sprite into separate frames. We can reverse this process, making the sprite show the whole image as a single picture by using `ClearSpriteAnimationFrames()`.

However, if all of the sprite's frames have been constructed using the `AddSpriteAnimationFrame()` statement, the sprite will have no image assigned to it after a `ClearSpriteAnimationFrames()` statement is executed.

The format for the `ClearSpriteAnimationFrames()` statement is shown in FIG-18.10.

FIG-18.10

`ClearSpriteAnimation
Frames()`

`ClearSpriteAnimationFrames` (`id`)

where:

id is an integer value giving the ID of the animated sprite.

SetSpriteActive()

A sprite can be made inactive. In this state, any animation will stop and also the physics of the sprite is also made inoperative (physics is covered in Chapter 20).

The `SetSpriteActive()` statement (see FIG-18.11) can cause a sprite to become inactive but is also used to reactivate the sprite.

FIG-18.11

`SetSpriteActive()`

`SetSpriteActive` (`id` , `istate`)

where:

id is an integer value giving the ID of the sprite.

istate is an integer value (0 or 1) used to set the state of the sprite (0: inactive; 1: active).

Activity 18.3

Modify project *Comet* so that, rather than play the explosion after 5 seconds, the sprite becomes inactive after 3 seconds.

Test and save your project.

GetSpriteActive()

The current state of a sprite can be determined using the `GetSpriteActive()` statement. This has the format shown in FIG-18.12.

FIG-18.12

GetSpriteActive()

integer `GetSpriteActive` (`id`)

where:

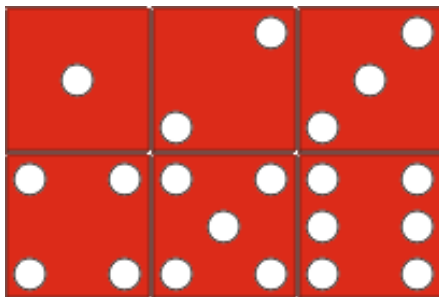
id is an integer value giving the ID of the sprite whose state is to be returned.

SetSpriteFrame()

Not all animated sprites are created with the intention of emulating movement; some are used to hold the various possible appearances of an object. For example, if we were using a sprite to represent a six sided dice, then we might use the image shown in FIG-18.13.

FIG-18.13

A Dice Image



With this type of image, we need to control which frame is showing at any time. Using the `PlaySprite()` statement would be inappropriate here. Instead, we can use `SetSpriteFrame()` which allows us to display any single frame within the animation.

The `SetSpriteFrame()` statement's format is shown in FIG-18.14.

FIG-18.14

SetSpriteFrame()

`SetSpriteFrame` (`id` , `iframe`)

where:

id is an integer value giving the ID of the animated sprite.

iframe is an integer value giving the number of the frame to be displayed. This should be in the range 1 to the number of frames in the animation.

The program in FIG-18.15 makes use of the dice image to show the value thrown by a six-sided dice. The value changes every 2 seconds.

FIG-18.15

Creating a Dice

```
rem *** Dice Sprite ***  
  
rem *** Load dice image ***  
LoadImage(1,"Dice.png",0)  
rem *** Create sprite ***  
CreateSprite(1,1)
```



FIG-18.15

(continued)

Creating a Dice

```

SetSpriteAnimation(1,68,68,6)
SetSpriteSize(1,10,-1)
SetSpritePosition(1,45,45)
rem *** Choose frame at random ***
SetSpriteFrame(1,Random(1,6))
rem *** record time ***
time = GetSeconds()
do
    rem *** If 2 secs passed, rethrow dice ***
    if GetSeconds() - time >= 2
        SetSpriteFrame(1,Random(1,6))
        time = GetSeconds()
    endif
    Sync()
loop

```

Activity 18.4

Start a new project called *DiceSprite* and implement the code given in FIG-18.15 (remember to copy the file *Dice.png* from *AGKDownloads/Chapter18*).

Test and save your project.

GetSpriteCurrentFrame()

To discover which frame of an animated sprite is currently being displayed, use the `GetSpriteCurrentFrame()` statement (see FIG-18.16).

FIG-18.16`GetSpriteCurrentFrame()`integer `GetSpriteCurrentFrame` (`id`)

where:

id is an integer value giving the ID of the animated sprite.

A Card Trick

Some sprites will be used to represent two-sided objects such as a playing card or coin. You can achieve the effect of turning over such an object by reducing the width of the object gradually, changing to the second frame, and then restoring the sprite's width. The technique is demonstrated in FIG-18.17.

FIG-18.17

Turning a Playing Card

```

rem *** Turning a card ***
rem *** Load image ***
LoadImage(1,"AceofClubs.png",0)
rem *** Create, size and place sprite ***
CreateSprite(1,1)
SetSpriteAnimation(1,256,387,2)
SetSpriteSize(1,20,-1)
SetSpritePosition(1,40,40)
rem *** Flip sprite ***
FlipCard(1)
rem *** Continue to flip ***
rem *** every 2 seconds ***
time = GetSeconds()

```



FIG-18.17

(continued)

Turning a Playing
Card

```

do
    if Getseconds() - time >= 2
        FlipCard(1)
        time = GetSeconds()
    endif
loop

function FlipCard(spr)
    rem *** Get the current dimensions of sprite ***
    spritewidth# = GetSpriteWidth(spr)
    spriteheight# = GetSpriteHeight(spr)
    rem *** Get the sprite's current position ***
    x# = GetSpriteX(spr)
    y# = GetSpriteY(spr)
    rem *** Set the reduction in width at each step ***
    stepsize# = 1.0
    rem *** Reduce sprite's width to zero ***
    for c# = spritewidth# to 0 step -stepsize#
        SetSpriteSize(1,c#,spriteheight#)
        rem *** Move sprite's left edge towards centre ***
        x# = x# + stepsize#/2
        SetSpritePosition(spr,x#,y#)
        Sync()
    next c#
    rem *** Swap to other frame in sprite ***
    SetSpriteFrame(spr,3-GetSpriteCurrentFrame(spr))
    rem *** Restore sprite's width ***
    for c# = 0 to spritewidth# step stepsize#
        SetSpriteSize(1,c#,spriteheight#)
        rem *** Move sprite's left edge away from centre ***
        x# = x# - stepsize#/2
        SetSpritePosition(spr,x#,y#)
        Sync()
    next c#
endfunction

```

Activity 18.5

Start a new project called *FlipCard* and implement the code given in FIG-18.17 (remember to copy the file *AceofClubs.png* from *AGKDownloads/Chapter18*).

Test and save your project.

The card flipping program turns the card using the lines

```

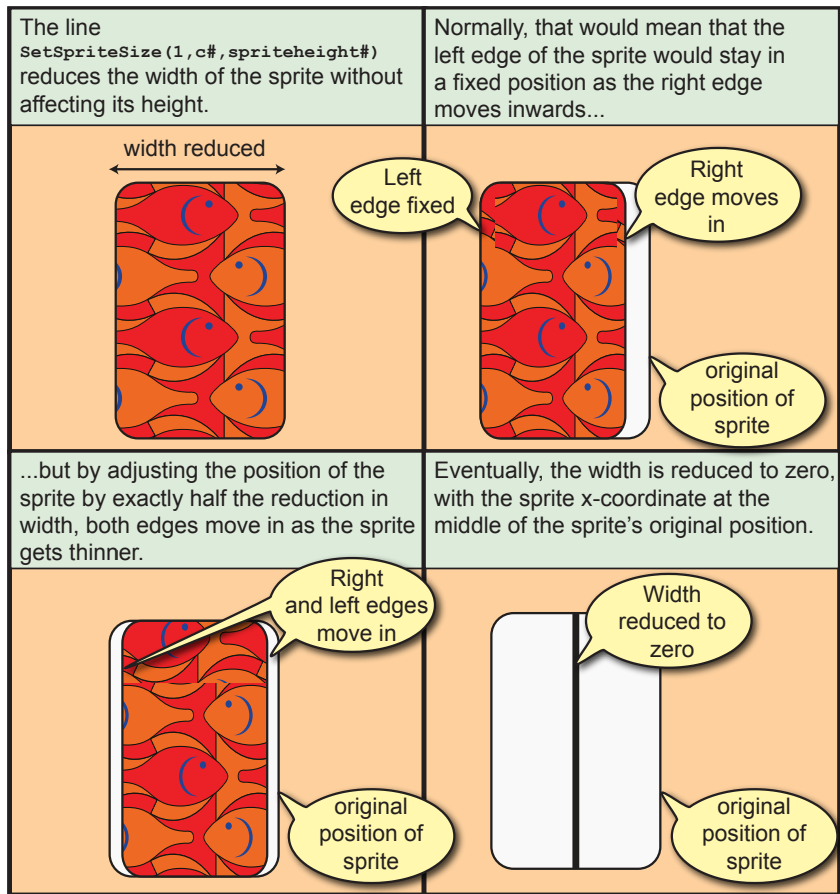
for c# = spritewidth# to 0 step -stepsize#
    SetSpriteSize(1,c#,spriteheight#)
    rem *** Move sprite's left edge towards centre ***
    x# = x# + stepsize#/2
    SetSpritePosition(spr,x#,y#)
    Sync()
next c#

```

The logic behind these lines is explained in FIG-18.18.

FIG-18.18

How the Card is
Turned



Once the second frame is displayed, the second **for** loop expands the sprite back to its original size, gradually moving the x-coordinate back to its starting value.

Summary

- An animated sprite is made up of a sequence of frames.
- A single image contains all the frames of an animated sprite.
- The image is loaded into a sprite in the normal way and then converted to an animated sprite using `SetSpriteAnimation()`.
- Use `PlaySprite()` to play the frames of an animated sprite.
- Use `AddSpriteAnimationFrame()` if you want to add new frames to an animation.
- Use `SetSpriteSpeed()` to adjust the speed at which the frames are changed.
- Use `StopSprite()` to halt an animation.
- Use `ResumeSprite()` to restart an animation from the point at which it stopped.
- Use `GetSpritePlaying()` to determine if an animated sprite is currently playing.

- Use `GetSpriteFrameCount()` to discover how many frames are in an animation.
- Use `ClearSpriteAnimationFrames()` to combine the frames of an animation into a single image which is displayed on the sprite.
- Use `SetSpriteActive()` to deactivate/activate a sprite. A deactivated sprite stops playing any associated animation and no longer reacts to physical forces.
- Use `GetSpriteActive()` to determine if a sprite is currently active.
- Use `SetSpriteFrame()` to make a sprite display a specific frame of its animation.
- Use `GetSpriteCurrentFrame()` to determine which frame is currently being displayed by an animated sprite.

An Asteroid Game

Introduction

This game implements the basic elements of an asteroid shooting game incorporating animated sprites and various other techniques that have been covered in the last few chapters.

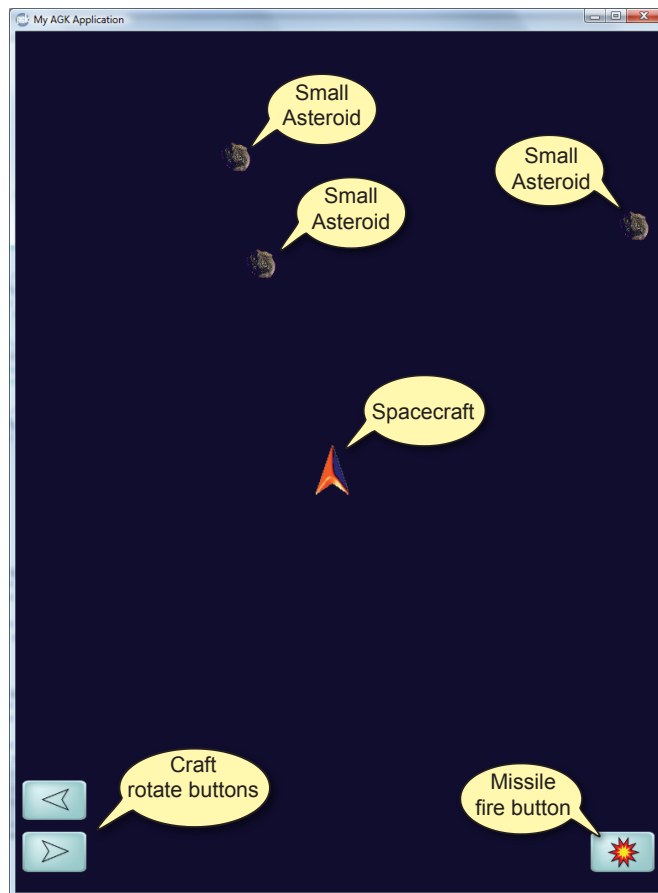
Since the main aim of the game is to demonstrate various design and programming ideas, the program itself lacks the finishing touches of a completed app in order to focus on the basic concepts. However, with very little effort, you could add the required elements to produce a complete game.

Game Layout

A labelled snapshot of the game is shown in FIG-18.19.

FIG-18.19

Asteroid Game Screen
Snapshot



The game begins with a single large asteroid travelling across the screen and a spacecraft at the centre of the screen. The controls allow the spacecraft to be rotated clockwise or counterclockwise and for the craft to fire missiles.

The snapshot in FIG-18.19 shows a stage in the game where the large asteroid has already been hit and been transformed into three smaller asteroids. When the smaller asteroids are hit by a missile, they will vanish from the screen.

Game Logic

The game uses the following logic:

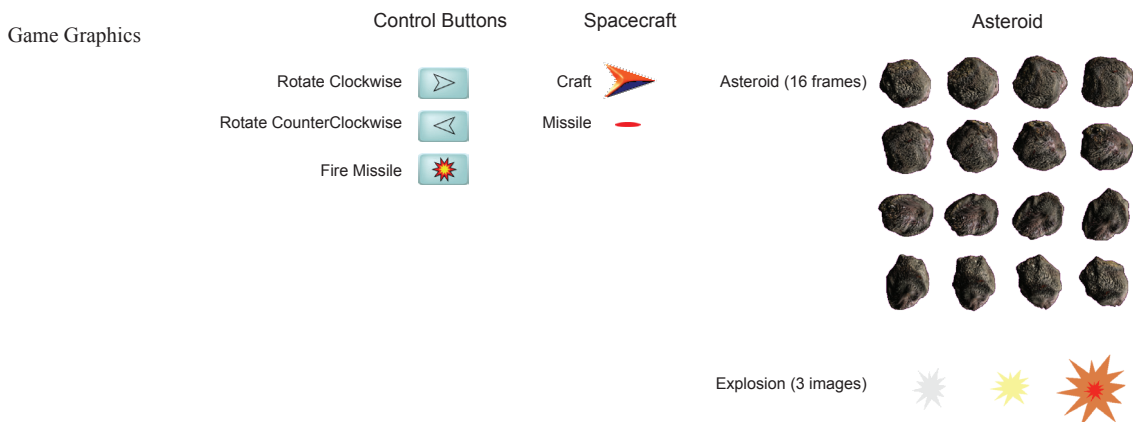
```
Load all resources used
Position the spacecraft
Position the controls
Create the largest asteroid
DO
    Move all asteroids
    Move all missiles
    React to ship controls
    Deal with collisions
LOOP
```

As usual, the overall logic is kept simple without going into too much detail at this early stage. Notice that there is no terminating condition. This makes testing easier, but later we could make the game stop when all asteroids are destroyed.

Game Resources

The images used by the game are shown in FIG-18.20.

FIG-18.20



Two sound files are also used. The first of these (*Launch.wav*) is played when a missile is launched and the second (*Explode.wav*) played when an asteroid is hit.

Game Code

In this project we will make use of the top-down programming method, creating code for the main game logic which will consist mostly of function calls and then implementing each function one at a time.

We need to start with any constants and global variables required by the program. The code for these is given below:

```
rem *** Constants ***
#constant shipID = 101
#constant leftButtonID = 102
#constant rightButtonID = 103
#constant fireButtonID = 104
#constant firstMissileID = 105
```

```

rem *** Record Structures ***
type MissileType
    xoffset as float
    yoffset as float
endtype

type AsteroidType
    size
    xoffset as float
    yoffset as float
endtype

rem *** Global variables ***
global highestAsteroidID = 0
global rotateSpeed# = 1.0
global lastDirection = 103
global numberOfMissiles = 0
global buttonUp = 1
global dim asteroids[4] as AsteroidType
global dim missiles[5] as MissileType

```

Being presented with these data items in this way is entirely artificial. If you were writing the program from scratch, then the need for each data item would only become clear as you created your code and so they would be defined as the coding went along. Here they have been presented to you simply to allow us to concentrate on implementing the code for the functions that are to follow. However, it is worth taking the time to give a brief description of each item.

The set of constants are, for the main part, the IDs to be assigned to the various sprites used within the program. However, the last constant is the ID of the first missile to be used. As we will see, the game allows up to five missiles to be on the screen at any one time, so these will be given the IDs 105 to 109. When a missile is destroyed, its ID can be reused by a later missile.

The *MissileType* structure defines the *x* and *y* offsets used to give a missile's velocity.

The *AsteroidType* structure not only gives the asteroid's velocity, but also the asteroid's type: large or small.

The global variables are used as follows:

<i>highestAsteroidID</i>	records the highest ID being used for an asteroid sprite.
<i>rotateSpeed#</i>	holds the angle by which the spacecraft rotates on each iteration of the main loop.
<i>lastDirection</i>	gives the ID of the last directional button pressed. This lets us know if the craft is turning clockwise or counterclockwise.
<i>numberOfMissiles</i>	is the number of missiles currently on the screen.
<i>buttonUp</i>	records the state of the fire button.
<i>asteroids</i> array	holds details of each asteroid.
<i>missiles</i> array	holds details of all missiles currently on screen.

Now we are ready to code the main program logic. This is taken almost unchanged from the outline logic of the game:

```
rem *** Main Game Logic ***
LoadResources()
PositionShip()
PositionControls()
CreateAsteroid(12,Random(0,100),Random(0,100))
do
    MoveAllAsteroids()
    ControlShip()
    MoveAllMissiles()
    HandleCollisions()
    Sync()
loop
end
```

The function names will give a clue to the purpose of each one. Only `CreateAsteroid()` requires parameters. These parameters give the width of the asteroid and where it is to be positioned on the screen.

Before we can compile our code, we need to create test stubs for each of the routines named in the main logic:

The code for the first test stub is

```
function LoadResources()
    Print("LoadResources()")
endfunction
```

The one for `CreateAsteroid()` needs to include parameters:

```
function CreateAsteroid(sz,x,y)
    Print("CreateAsteroid()")
endfunction
```

Activity 18.6

Create a new project called *Asteroids*. Set the size of the app window to 768 wide by 1024 high.

Compile the default code and copy the following files from *AGKDownloads/Chapter18* to the project's *media* folder.

<i>Arrow.png</i>	<i>Asteroid.png</i>	<i>Explode01.png</i>
<i>Explode02.png</i>	<i>Explode03.png</i>	<i>Fire.png</i>
<i>Left.png</i>	<i>Missile.png</i>	<i>Right.png</i>
<i>Explode.wav</i>	<i>Launch.wav</i>	

Replace the default code with the code containing details of the program's constants, global variables, etc.

Add the code for the main program logic as shown above.

Enter the test stubs for `LoadResources()` and `CreateAsteroid()`. Add test stubs for the remaining routines called by the main logic.

Run the program. What text is displayed? Save the project.

Now that the test stubs are in place, we can begin to replace each with the actual code they should contain.

LoadResources()

This routine loads all the image and sound files required by the program. Its code is:

```
function LoadResources()  
    rem *** Asteroid images ***  
    LoadImage(1,"Asteroid.png")  
    LoadImage(2,"Explode01.png")  
    LoadImage(3,"Explode02.png")  
    LoadImage(4,"Explode03.png")  
    rem *** Ship and missile images ***  
    LoadImage(5,"Arrow.png")  
    LoadImage(6,"Missile.png")  
    rem *** Button images ***  
    LoadImage(7,"Left.png")  
    LoadImage(8,"Right.png")  
    LoadImage(9,"Fire.png")  
    rem *** Sound files ***  
    LoadSound(1,"Launch.wav")  
    LoadSound(2,"Explode.wav")  
endfunction
```

Activity 18.7

Update *Asteroids*, replacing the test stub for `LoadResources()` with the routine's final code (as shown above).

Run the project. If any of the files are missing from the *media* folder or you have misspelled a file name, a runtime error will occur.

Resave your project.

PositionShip()

This function creates a spacecraft sprite using the *Arrow.png* image. The sprite is 5% wide and its top-left corner should be placed at position (47.5,48) and rotated to -90°.

Activity 18.8

Update *Asteroids*, replacing the test stub for `PositionShip()` with the routine's final code (which you need to create).

Test your code. If it is correct, the spacecraft should appear at the centre of the screen.

Resave your project.

PositionControls()

This function positions the three buttons used to control the ship. On the left side of the screen are the rotate-right and rotate-left buttons which allow the ship to be rotated. On the right side of the screen is the fire button which fires a missile from the spacecraft.

The buttons make use of the images with ID codes 7, 8 and 9.

Activity 18.9

Add the code for `PositionControls()` to *Asteroids*.

All buttons are 10% wide and should be placed on layer 9.

Button positions are:

left button	at	(1,87)
right button	at	(1,93)
fire button	at	(89,93)

Test your code. If it is correct, the three buttons should appear on the screen.

Resave your project.

CreateAsteroid()

This function creates an animated asteroid sprite. Its mini-spec is given below:

FUNCTION NAME	:	CreateAsteroid
PARAMETERS		
In	:	sz : Integer x : Integer y : Integer
GLOBALS		
Written	:	highestSpriteID asteroids[]
PRE-CONDITION	:	None
DESCRIPTION	:	Increment <i>highestSpriteID</i> Create sprite with ID of <i>highestSpriteID</i> using image 1 Set sprite width to <i>sz</i> (height: -1) Position sprite at (x,y) Convert sprite to an animated sprite with 16 frames (frames are 250x250 pixels) Add images 2, 3 and 4 to the animation as frames 17 to 19 Set <i>asteroids[highestSpriteID].size</i> to <i>sz</i> Set <i>asteroids[highestAsteroidID].xoffset</i> to a random value between -1 and 1 (step size of 0.1) Set <i>asteroids[highestAsteroidID].yoffset</i> to a random value between -1 and 1 (step size of 0.1) Play the animation repeatedly (frames 1 to 16 only at a frame rate of 20 fps)

Activity 18.10

Implement the `CreateAsteroid()` function and add it to *Asteroids*.

Run the project. *A rotating asteroid should appear on the screen.*

Resave your project.

MoveAllAsteroids()

This function moves any existing asteroids using the values held in the *asteroids* array. The first, large asteroid has an ID of 1; the three smaller ones which are created when the larger one is hit by a missile, have IDs 2, 3, and 4. This routine cycles through the four possible IDs to check if a sprite with that ID exists. If it does, then a new function called `MoveSingleAsteroid()` is called with the sprite ID as a parameter.

The logic of this routine can be described in structured English as:

```
FOR each possible asteroid sprite ID DO
  IF a sprite of that ID exists THEN
    Move the asteroid of that ID
  ENDIF
ENDFOR
```

Activity 18.11

Implement the `MoveAllAsteroids()` function and add it to *Asteroids*.

Add a test stub for the new routine `MoveSingleAsteroid()` to the program.

Run the project. *A text message should be displayed saying `MoveSingleAsteroid()` has been called.*

Resave your project.

MoveSingleAsteroid()

This function moves the asteroid of a specified ID by the appropriate offset value stored in the *asteroids* array. The code for the routine is:

```
function MoveSingleAsteroid(id)
  rem *** Calculate new coordinates ***
  x# = GetSpriteX(id) + asteroids[id].xoffset
  y# = GetSpriteY(id) + asteroids[id].yoffset
  rem *** If it is past an edge, bring it ***
  rem *** back on at the opposite edge ***
  if x# > 120
    x# = 0
  elseif x# < -20
    x# = 100
  elseif y# > 120
    y# = 0
  elseif y# < -20
    y# = 100
  endif
  rem *** Reposition sprite ***
  SetSpritePosition(id,x#,y#)
endfunction
```

Notice that it waits until the asteroid is 20% off the screen before bringing it round to the opposite edge. This is to allow the player time to think before the asteroid reappears. You may also have noticed that no attempt is made to have the asteroid follow the same path each time it crosses the screen. If it were to do this, then the player's task of predicting the position of the asteroid would be too easy.

Methods of making a sprite reappear at the opposite edge of the screen were discussed in Chapter 17.

Activity 18.12

Implement the `MoveSingleAsteroid()` function and add it to *Asteroids*.

Run the project. *The asteroid should now move across the screen, reappearing at the opposite edge when it moves off the screen.*

Resave your project.

ControlShip()

This function allows the player to use the screen button to control the rotation of the ship and to fire missiles. A maximum of five missiles can be on the screen at one time.

This routine has the following code:

```
function ControlShip()
    rem *** Check for button pressed down ***
    x = GetPointerX()
    y = GetPointerY()
    id = GetSpriteHit(x,y)
    rem *** If button is pressed ***
    if GetPointerState() = 1
        rem *** Handle button ***
        select id
            case leftButtonID: //Rotate ship left
                SetSpriteAngle(shipID,GetSpriteAngle(shipID) -
                    ↵rotatespeed#)
            endcase
            case rightButtonID: //Rotate ship right
                SetSpriteAngle(shipID,GetSpriteAngle(shipID) +
                    ↵rotatespeed#)
            endcase
            case fireButtonID: //Handle missile firing
                rem *** If less than 5 missiles on screen ***
                rem *** and fire button has just been pressed ***
                if numberOfMissiles < 5 and buttonup = 1
                    rem *** Create a new missile ***
                    CreateMissile()
                    rem *** Record fire button as pressed ***
                    buttonup = 0
                endif
            endcase
        endselect
    else
        rem *** If no buttons are pressed ***
        rem *** record fire button as up ***
        buttonup = 1
    endif
endfunction
```

In fact, this routine will return the ID of any sprite being pressed - button, asteroid, ship or missile (see first three lines) - but only the buttons are processed by the `select` statement, so hits on any of the sprites would be ignored.

The left and right buttons result in a change to the ship's rotation setting. By how much this is changed is determined by the setting of the constant `rotatespeed#`.

The handling of a missile launch is slightly more complicated. Because this routine will be executed several times per second (it is called from within the main logic's `do..loop` stucture) all five missiles would be launched in the time it takes us mere humans to press and release the **fire** button. To avoid this, the global variable *buttonup* is used to determine the state of the **fire** button, ensuring that only one missile can be launched on each press.

Initially, *buttonup* is set to 1 (indicating that the **fire** button is unpressed). When the **fire** button is pressed by the player, the code checks to see that the *buttonup* variable is set to 1 before allowing a missile to be created. When the missile is created *buttonup* is then set to zero thereby making the immediate launch of another missile impossible. *buttonup* is only reset to 1 when no buttons are being pressed.

The other check used when creating a missile is that a maximum of five missiles can exist at any one time. Each time a missile is created, the global variable *numberOfMissiles* is incremented; when a missile is destroyed, the variable is decremented.

Activity 18.13

Implement the `ControlShip()` function and add it to *Asteroids*.

Add a test stub for the new routine `CreateMissile()` to the program.

Run the project. *The ship should rotate clockwise or counterclockwise when the **right** or **left** buttons are pressed. Pressing the **fire** button should display a message saying `CreateMissile()` has been called.*

Resave your project.

CreateMissile()

This function creates a missile whose orientation matches that of the ship. The missile is placed at the centre of the ship, but on layer 11 so it is hidden by the ship. The code for the function is:

```
function CreateMissile()
  rem *** Check pre-condition ***
  rem *** Exit if 5 missiles already exist ***
  if numberOfMissiles >= 5
    exitfunction
  endif
  rem *** Increment missile count ***
  inc numberOfMissiles
  rem For each missile ID available ***
  for c = firstMissileID to firstMissileID + 4
    rem *** If no missile of that ID ***
    if GetSpriteExists(c) = 0
      rem *** Create the missile under ship ***
      CreateSprite(c,6)
      SetSpriteDepth(c,11)
      SetSpriteSize(c,2,-1)
      SetSpritePositionByOffset(c,
        ↵GetSpriteXByOffset(shipID),
        ↵GetSpriteYByOffset(shipID))
      rem *** Rotate it to same angle as ship ***
      angle# = GetSpriteAngle(shipID)
```

```

        SetSpriteAngle(c,angle#)
rem *** Set missiles velocity ***
missiles[c-firstMissileID].xoffset = sin(angle#)*2
missiles[c-firstMissileID].yoffset =-cos(angle#)*2
rem *** Play the launch sound ***
PlaySound(1)
rem *** Exit FOR loop ***
exit
    endif
next c
endfunction

```

Notice that the function checks to see if five missiles already exist and exits the routine if they do. Now, you may be tempted to think that since we did the same check before calling the routine, there is no need to do the same check here. However, it is always good policy for a routine's code not to make any assumptions about what safeguards will be in place when that function is being called; hence the duplication of the check.

Activity 18.14

Implement the `CreateMissile()` function and add it to *Asteroids*.

Run the project. *When you press the fire button, the missile is placed "under" the ship and will not be visible.*

Temporarily change the missile's layer setting to 9 and check that it has been correctly positioned. *The launch sound will play when a missile is created. Only five missiles can be created.*

Resave your project.

MoveAllMissiles()

This function tests each ID available to the missiles' sprites (*firstMissileID* to *firstMissileID* + 4) to discover if a sprite of that ID exists. If the sprite does exist, a new function called `MoveSingleMissile()` is called to move that particular missile. `MoveSingleMissile()` takes the ID of the missile sprite as a parameter.

Activity 18.15

Implement the `MoveAllMissiles()` function based on the description given above (HINT: This is a very short routine, containing only five lines of code) and add it to *Asteroids*. Add a test stub for the new routine `MoveSingleMissile()` to the program.

Run the project. *When you press the fire button, a text message stating that the function `MoveSingleMissile()` has been called will be displayed.*

Resave your project.

MoveSingleMissile()

This function moves the missile of a specified ID by the appropriate offset value stored in the *missiles* array. If the missile moves off the edge of the screen, it is deleted and the missile count held in *numberOfMissiles* is decremented. This allows a new missile to be fired. The code for the routine is:

```

function MoveSingleMissile(id)
    rem *** If missile doesn't exist ***
    rem *** exit function ***
    if GetSpriteExists(id) = 0
        exitfunction
    endif
    rem *** Move the missile ***
    SetSpritePosition(id,GetSpriteX(id)+
    ↵missiles[id-firstMissileID].xoffset,GetSpriteY(id)+
    ↵missiles[id-firstMissileID].yoffset)
    rem *** If missile leaves the screen ***
    rem *** delete it and decrease missile count ***
    if GetSpriteX(id) < 0 or GetSpriteX(id) > 105 or
    ↵GetSpriteY(id) < 0 or GetSpriteY(id) > 105
        DeleteSprite(id)
        dec numberOfMissiles
    endif
endfunction

```

Activity 18.16

Implement the `MoveSingleMissiles()` function and add it to *Asteroids*.

Run the project. *When you press the fire button, the missile should move in the direction of the spacecraft.*

Resave your project.

HandleCollisions()

At this point, the only collisions we want to detect are those between a missile and an asteroid. If a missile hits a large asteroid, the missile is destroyed and the asteroid breaks into three smaller asteroids. If a smaller asteroid is hit, both the missile and asteroid are destroyed. An explosion sound is played when an asteroid is hit.

The function's code checks each possible missile ID against each asteroid ID to test for a collision. The function's code is:

```

function HandleCollisions()
    rem *** FOR each missile ID DO ***
    for m = firstMissileID to firstMissileID + 4
        rem *** If the missile exists check each asteroid ***
        if GetSpriteExists(m) = 1
            rem *** FOR each asteroid ID DO ***
            for a = 1 to highestAsteroidID
                rem *** If asteroid exists ***
                if GetSpriteExists(a)=1
                    rem *** If missile and asteroid have
                    ↵collided ***
                    if GetSpriteCollision(m,a) = 1
                        rem *** Delete missile and reduce
                        ↵missile count ***
                        DeleteSprite(m)
                        dec numberOfMissiles
                        rem *** If a large asteroid split it ***
                        if asteroids[a].size = 12
                            SplitAsteroid(a)
                        else
                            rem *** if a small asteroid, destroy
                            ↵it ***

```



```

        DestroyAsteroid(a)
    endif
    rem *** Don't test remaining asteroids
    exit
endif
endif
next a
endif
next m
endfunction

```

Two new functions are called from within the code. These are `SplitAsteroid()` and `DestroyAsteroid()`.

Activity 18.17

Implement the `HandleCollisions()` function and add it to *Asteroids*.

Add test stubs for the new routine `SplitAsteroid()` and `DestroyAsteroid()` to the program.

Run the project. *When a missile hits the asteroid, you should see text stating that the `SplitAsteroid()` function has been executed.*

Resave your project.

SplitAsteroid()

This function splits the specified asteroid sprite into three smaller asteroid sprites placing them at the same position as the original one. The original sprite is then destroyed. The mini-spec for this function is:

FUNCTION NAME	:	SplitAsteroid
PARAMETERS		
In	:	id : Integer
PRE-CONDITION	:	Sprite with ID <i>id</i> exists
DESCRIPTION	:	Find the coordinates of the specified sprite Create three new sprites (width: 6) at this position Delete sprite <i>id</i>

Activity 18.18

Implement the `SplitAsteroid()` function and add it to *Asteroids*. (HINT: The new asteroids are created by three calls to `CreateAsteroid()`.)

Run the project. *When a missile hits the asteroid, it should split into three smaller asteroids but the large asteroid will remain.*

Resave your project.

DestroyAsteroid()

The final routine deletes the asteroid of the specified ID. The sprite to be destroyed first plays the last three frames of the animation (frames 17 to 19) which show the

explosion. This is followed by the explosion sound and, finally, the sprite is deleted.

The code for this function is:

```
function DestroyAsteroid(id)
    rem *** Play the explosion frames ***
    PlaySprite(id,10,1,17,19)
    Sync()
    rem *** Play explosion noise ***
    PlaySound(2)
    rem *** Delete sprite ***
    DeleteSprite(id)
endfunction
```

Activity 18.19

Implement the `DestroyAsteroid()` function and add it to *Asteroids*.

Test your completed project.

Reorganise the layout of your code, grouping Level 1 and Level 2 functions separately.

Save your project.

Of course, much more can be done to turn this into a complete game. The most obvious need is to check for a collision between an asteroid and the spacecraft. But you would also need to add more large asteroids at the start of the game, display a score, etc.

Solutions

Activity 18.1

Code for *Comet*:

```
rem *** Animated Sprite ***
rem *** load image ***
LoadImage(1,"Asteroid.png")
rem *** Create, size and position sprite ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
SetSpritePosition(1,46,46)
rem *** Transform into animated sprite ***
SetSpriteAnimation(1,250,250,16)
rem *** Play animation ***
PlaySprite(1)
do
    Sync()
loop
```

To play at 20 frames per second, change the `PlaySprite()` command to

```
PlaySprite(1,20)
```

Activity 18.2

Modified code for *Comet* (additional frames added):

```
rem *** Animated Sprite ***
rem *** Load images ***
LoadImage(1,"Asteroid.png")
LoadImage(2,"Explode01.png")
LoadImage(3,"Explode02.png")
LoadImage(4,"Explode03.png")
rem *** Create, size and position sprite ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
SetSpritePosition(1,46,46)
rem *** Transform into animated sprite ***
SetSpriteAnimation(1,250,250,16)
rem *** Add new frames to the animation ***
AddSpriteAnimationFrame(1,2)
AddSpriteAnimationFrame(1,3)
AddSpriteAnimationFrame(1,4)
rem *** Play animation ***
PlaySprite(1,20)
do
    Sync()
loop
```

To play only frames 1 to 16, change the line

```
PlaySprite(1,20)
```

to

```
PlaySprite(1,20,1,1,16)
```

Modified code for *Comet* (explosion after 5 seconds):

```
rem *** Animated Sprite ***
rem *** Load images ***
LoadImage(1,"Asteroid.png")
LoadImage(2,"Explode01.png")
LoadImage(3,"Explode02.png")
LoadImage(4,"Explode03.png")
rem *** Create, size and position sprite ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
SetSpritePosition(1,46,46)
rem *** Transform into animated sprite ***
SetSpriteAnimation(1,250,250,16)
rem *** Add new frames to the animation ***
AddSpriteAnimationFrame(1,2)
AddSpriteAnimationFrame(1,3)
AddSpriteAnimationFrame(1,4)
rem *** Play animation ***
PlaySprite(1,20,1,1,16)
rem *** Explode after 5 seconds ***
time = GetSeconds()
```

```
repeat
```

```
    Sync()
until GetSeconds() - time >= 5
PlaySprite(1,20,0,17,19)
do
    Sync()
loop
```

Activity 18.3

Modified code for *Comet* (inactive after 3 secs) :

```
rem *** Animated Sprite ***
rem *** Load images ***
LoadImage(1,"Asteroid.png")
LoadImage(2,"Explode01.png")
LoadImage(3,"Explode02.png")
LoadImage(4,"Explode03.png")
rem *** Create, size and position sprite ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
SetSpritePosition(1,46,46)
rem *** Transform into animated sprite ***
SetSpriteAnimation(1,250,250,16)
rem *** Add new frames to the animation ***
AddSpriteAnimationFrame(1,2)
AddSpriteAnimationFrame(1,3)
AddSpriteAnimationFrame(1,4)
rem *** Play animation ***
PlaySprite(1,20,1,1,16)
rem *** After 3 secs make sprite inactive ***
time = GetSeconds()
repeat
    Sync()
until GetSeconds() - time >= 3
SetSpriteActive(1,0)
do
    Sync()
loop
```

Activity 18.4

No solution required.

Activity 18.5

No solution required.

Activity 18.6

Code for *Asteroids*:

```
rem *** Asteroid Game ***

rem *** Constants ***
#constant shipID = 101
#constant leftButtonID = 102
#constant rightButtonID = 103
#constant fireButtonID = 104
#constant firstMissileID = 105
rem *** Record Structures ***
type MissileType
    xoffset as float
    yoffset as float
endtype

type AsteroidType
    size
    xoffset as float
    yoffset as float
endtype

rem *** Global variables ***
global highestAsteroidID = 0
global rotateSpeed# = 1.0
global lastDirection = 103
global numberOfMissiles = 0
global buttonUp = 1
global dim asteroids[4] as AsteroidType
global dim missiles[5] as MissileType

rem *** Main Game Logic ***
LoadResources()
PositionShip()
```

```

PositionControls ()
CreateAsteroid(12,Random(0,100),Random(0,100))
do
    MoveAllAsteroids()
    ControlShip()
    MoveAllMissiles()
    HandleCollisions()
    Sync()
loop
end

function LoadResources ()
    Print("LoadResources() ")
endfunction

function PositionShip()
    Print("PositionShip() ")
endfunction

function PositionControls()
    Print("PositionControls() ")
endfunction

function CreateAsteroid(sz,x,y)
    Print("CreateAsteroid() ")
endfunction

function MoveAllAsteroids()
    Print("MoveAllAsteroids() ")
endfunction

function ControlShip()
    Print("ControlShip() ")
endfunction

function MoveAllMissiles()
    Print("MoveAllMissiles() ")
endfunction

function HandleCollisions()
    Print("HandleCollisions() ")
endfunction

```

The screen will display the names of all eight routines, but the first four will be shown only once (and you are unlikely to see them because of the next call to `Sync()`). The names of the four functions called within the `do...loop` will remain on the screen.

Activity 18.7

Modified code for *Asteroids*(`LoadResources()` coded):

```

rem *** Asteroid Game ***

rem *** Constants ***
#constant shipID = 101
#constant leftButtonID = 102
#constant rightButtonID = 103
#constant fireButtonID = 104
#constant firstMissileID = 105
rem *** Record Structures ***
type MissileType
    xoffset as float
    yoffset as float
endtype

type AsteroidType
    size
    xoffset as float
    yoffset as float
endtype

rem *** Global variables ***
global highestAsteroidID = 0
global rotateSpeed# = 1.0
global lastDirection = 103
global numberOfMissiles = 0
global buttonUp = 1
global dim asteroids[4] as AsteroidType
global dim missiles[5] as MissileType

rem *** Main Game Logic ***
LoadResources()
PositionShip()

```

```

PositionControls ()
CreateAsteroid(12,Random(0,100),Random(0,100))
do
    MoveAllAsteroids()
    ControlShip()
    MoveAllMissiles()
    HandleCollisions()
    Sync()
loop
end

function LoadResources ()
    rem *** Asteroid images ***
    LoadImage(1,"Asteroid.png")
    LoadImage(2,"Explode01.png")
    LoadImage(3,"Explode02.png")
    LoadImage(4,"Explode03.png")
    rem *** Ship and missile images ***
    LoadImage(5,"Arrow.png")
    LoadImage(6,"Missile.png")
    rem *** Button images ***
    LoadImage(7,"Left.png")
    LoadImage(8,"Right.png")
    LoadImage(9,"Fire.png")
    rem *** Sound files ***
    LoadSound(1,"Launch.wav")
    LoadSound(2,"Explode.wav")
endfunction

function PositionShip()
    Print("PositionShip() ")
endfunction

function PositionControls()
    Print("PositionControls() ")
endfunction

function CreateAsteroid(sz,x,y)
    Print("CreateAsteroid() ")
endfunction

function MoveAllAsteroids()
    Print("MoveAllAsteroids() ")
endfunction

function ControlShip()
    Print("ControlShip() ")
endfunction

function MoveAllMissiles()
    Print("MoveAllMissiles() ")
endfunction

function HandleCollisions()
    Print("HandleCollisions() ")
endfunction

```

Activity 18.8

Modified code for *Asteroids*(`PositionShip()` coded):

```

rem *** Asteroid Game ***

rem *** Constants ***
#constant shipID = 101
#constant leftButtonID = 102
#constant rightButtonID = 103
#constant fireButtonID = 104
#constant firstMissileID = 105
rem *** Record Structures ***
type MissileType
    xoffset as float
    yoffset as float
endtype

type AsteroidType
    size
    xoffset as float
    yoffset as float
endtype

rem *** Global variables ***
global highestAsteroidID = 0
global rotateSpeed# = 1.0
global lastDirection = 103
global numberOfMissiles = 0
global buttonUp = 1

```

```
global dim asteroids[4] as AsteroidType
global dim missiles[5] as MissileType
```

```
rem *** Main Game Logic ***
LoadResources()
PositionShip()
PositionControls()
CreateAsteroid(12,Random(0,100),Random(0,100))
do
    MoveAllAsteroids()
    ControlShip()
    MoveAllMissiles()
    HandleCollisions()
    Sync()
loop
end

function LoadResources()
    rem *** Asteroid images ***
    LoadImage(1,"Asteroid.png")
    LoadImage(2,"Explode01.png")
    LoadImage(3,"Explode02.png")
    LoadImage(4,"Explode03.png")
    rem *** Ship and missile images ***
    LoadImage(5,"Arrow.png")
    LoadImage(6,"Missile.png")
    rem *** Button images ***
    LoadImage(7,"Left.png")
    LoadImage(8,"Right.png")
    LoadImage(9,"Fire.png")
    rem *** Sound files ***
    LoadSound(1,"Launch.wav")
    LoadSound(2,"Explode.wav")
endfunction

function PositionShip()
    rem *** Create, size and position spacecraft ***
    CreateSprite(shipID,5)
    SetSpriteSize(shipID,5,-1)
    SetSpritePosition(shipID,47.5,48)
    rem *** Rotate ship to point upwards ***
    SetSpriteAngle(shipID,-90)
endfunction

function PositionControls()
    Print("PositionControls() ")
endfunction

function CreateAsteroid(sz,x,y)
    Print("CreateAsteroid() ")
endfunction

function MoveAllAsteroids()
    Print("MoveAllAsteroids() ")
endfunction

function ControlShip()
    Print("ControlShip() ")
endfunction

function MoveAllMissiles()
    Print("MoveAllMissiles() ")
endfunction

function HandleCollisions()
    Print("HandleCollisions() ")
endfunction
```

Activity 18.9

Code for *Asteroids*' PositionControls():

```
function PositionControls()
    rem *** Position Left button ***
    CreateSprite(leftButtonID,7)
    SetSpriteSize(leftButtonID,-1)
    SetSpritePosition(leftButtonID,1,87)
    SetSpriteDepth(leftButtonID,9)
    rem *** Position right button ***
    CreateSprite(rightButtonID,8)
    SetSpriteSize(rightButtonID,-1)
    SetSpritePosition(rightButtonID,1,93)
    SetSpriteDepth(rightButtonID,9)
    rem *** Position fire button ***
    CreateSprite(fireButtonID,9)
    SetSpriteSize(fireButtonID,-1)
    SetSpritePosition(fireButtonID,89,93)
    SetSpriteDepth(fireButtonID,9)
```

```
endfunction
```

Activity 18.10

Code for *Asteroids*' CreateAsteroid():

```
function CreateAsteroid(sz,x,y)
    inc highestAsteroidID
    id = highestAsteroidID
    rem *** Create, size and position sprite ***
    CreateSprite(id,1)
    SetSpriteSize(id,sz,-1)
    SetSpritePosition(id,x,y)
    rem *** Transform into animated sprite ***
    SetSpriteAnimation(id,250,250,16)
    AddSpriteAnimationFrame(id,2)
    AddSpriteAnimationFrame(id,3)
    AddSpriteAnimationFrame(id,4)
    rem *** Record sprite's size and velocity ***
    asteroids[id].size = sz
    asteroids[id].xoffset = (Random(0,20)-10)/10.0
    asteroids[id].yoffset = (Random(0,20)-10)/10.0
    PlaySprite(id,20,1,1,16)
endfunction
```

Activity 18.11

Code for *Asteroids*' MoveAllAsteroids():

```
function MoveAllAsteroids()
    for c = 1 to highestAsteroidID
        if GetSpriteExists(c) = 1
            MoveSingleAsteroid(c)
        endif
    next c
endfunction
```

Test stub for *MoveSingleAsteroid()*:

```
function MoveSingleAsteroid(id)
    Print("MoveSingleAsteroid() ")
endfunction
```

Activity 18.12

Code for *Asteroids*' MoveSingleAsteroid():

```
function MoveSingleAsteroid(id)
    rem *** Calculate new coordinates ***
    x# = GetSpriteX(id) + asteroids[id].xoffset
    y# = GetSpriteY(id) + asteroids[id].yoffset
    rem *** If it is past an edge, bring it ***
    rem *** back on at the opposite edge ***
    if x# > 120
        x# = 0
    elseif x# < -20
        x# = 100
    elseif y# > 120
        y# = 0
    elseif y# < -20
        y# = 100
    endif
    rem *** Reposition sprite ***
    SetSpritePosition(id,x#,y#)
endfunction
```

Activity 18.13

Code for *Asteroids*' ControlShip():

```
function ControlShip()
    rem *** Check for button pressed down ***
    x = GetPointerX()
    y = GetPointerY()
    id = GetSpriteHit(x,y)
    rem *** If button is pressed ***
    if GetPointerState() = 1
        rem *** Handle button ***
        select id
            case leftButtonID: //Rotate ship left
                SetSpriteAngle(shipID,GetSpriteAngle(
                    shipID)-rotatespeed#)
            endcase
            case rightButtonID: //Rotate ship right
                SetSpriteAngle(shipID,
                    GetSpriteAngle(shipID)+
                    rotatespeed#)
```

```

        endcase
    case fireButtonID: //Handle missile firing
        rem *** If less than 5 missiles on
        ↵screen ***
        rem *** and fire button has just been
        ↵pressed ***
        if numberOfMissiles<5 and buttonup=1
            rem *** Create a new missile ***
            CreateMissile()
            rem *** Record fire button as
            ↵being pressed ***
            buttonup = 0
        endif
    endcase
endselect
else
    rem *** If no buttons are pressed ***
    rem *** record fire button as up ***
    buttonup = 1
endif
endfunction

```

Test stub for *CreateMissile()*:

```

function CreateMissile()
    Print("CreateMissile()")
endfunction

```

Activity 18.14

Code for *Asteroids'* createMissile():

```

function CreateMissile()
    rem *** Check pre-condition ***
    rem *** Exit if 5 missiles already exist ***
    if numberOfMissiles >= 5
        exitfunction
    endif
    rem *** Increment missile count ***
    inc numberOfMissiles
    rem For each missile ID available ***
    for c = firstMissileID to firstMissileID + 4
        rem *** If no missile of that ID ***
        if GetSpriteExists(c) = 0
            rem *** Create the missile under ship
            ↵***
            CreateSprite(c,6)
            SetSpriteDepth(c,11)
            SetSpriteSize(c,2,-1)
            SetSpritePositionByOffset(c,
            ↵GetSpriteXByOffset(shipID),
            ↵GetSpriteYByOffset(shipID))
            rem *** Rotate it to same angle as ship
            ↵***
            angle# = GetSpriteAngle(shipID)
            SetSpriteAngle(c,angle#)
            rem *** Set missiles velocity ***
            missiles[c-firstMissileID].xoffset =
            ↵cos(angle#)*2
            missiles[c-firstMissileID].yoffset =
            ↵sin(angle#)*2
            rem *** Play the launch sound ***
            PlaySound(1)
            rem *** Exit the FOR loop ***
            exit
        endif
    next c
endfunction

```

To have the missile appear over the ship change the line

```
SetSpriteDepth(c,11)
```

to

```
SetSpriteDepth(c,9)
```

Return to the original version of the line after your test run.

Activity 18.15

Code for *Asteroids'* MoveAllMissiles():

```

function MoveAllMissiles()
    for c = firstMissileID to firstMissileID + 4
        if GetSpriteExists(c) = 1
            MoveSingleMissile(c)
        endif
    next c
endfunction

```

```

        next c
    endfunction

```

Test stub for *MoveSingleMissile()*:

```

function MoveSingleMissile()
    Print("MoveSingleMissile()")
endfunction

```

Activity 18.16

Code for *Asteroids'* MoveSingleMissile():

```

function MoveSingleMissile(id)
    rem *** If missile doesn't exist ***
    rem *** exit function ***
    if GetSpriteExists(id) = 0
        exitfunction
    endif
    rem *** Move the missile ***
    SetSpritePosition(id,GetSpriteX(id)+
    ↵missiles[id-firstMissileID].xoffset,
    ↵GetSpriteY(id)+missiles[id-firstMissileID].
    ↵yoffset)
    rem *** If missile leaves the screen ***
    rem *** delete it and decrease missile count ***
    if GetSpriteX(id) < 0 or GetSpriteX(id) > 105 or
    ↵GetSpriteY(id) < 0 or GetSpriteY(id) > 105
        DeleteSprite(id)
        dec numberOfMissiles
    endif
endfunction

```

Activity 18.17

Code for *Asteroids'* HandleCollisions():

```

function HandleCollisions()
    rem *** FOR each missile ID DO ***
    for m = firstMissileID to firstMissileID + 4
        rem *** If the missile exists ***
        if GetSpriteExists(m) = 1
            rem *** FOR each asteroid ID DO ***
            for a = 1 to highestAsteroidID
                rem *** If asteroid exists ***
                if GetSpriteExists(a)=1
                    rem *** If missile and asteroid
                    ↵have collided ***
                    if GetSpriteCollision(m,a) = 1
                        rem *** Delete missile and
                        ↵reduce missile count ***
                        DeleteSprite(m)
                        dec numberOfMissiles
                        rem *** If its a large
                        ↵asteroid split the asteroid
                        ↵***
                        if asteroids[a].size = 12
                            SplitAsteroid(a)
                        else
                            rem *** if its a small
                            ↵asteroid, destroy it
                            DestroyAsteroid(a)
                        endif
                        exit
                    endif
                endif
            next a
        endif
    next m
endfunction

```

Test stub for *SplitAsteroid()*:

```

function SplitAsteroid()
    Print("SplitAsteroid()")
endfunction

```

Test stub for *DestroyAsteroid()*:

```

function DestroyAsteroid()
    Print("DestroyAsteroid()")
endfunction

```

Activity 18.18

Code for *Asteroids'* SplitAsteroid():

```

function SplitAsteroid(id)

```

```

rem *** If sprite doesn't exist, exit ***
if GetSpriteExists(id) = 0
    exitfunction
endif
rem *** Get the asteroid's current position ***
x = GetSpriteX(id)
y = GetSpriteY(id)
rem *** Create three new asteroids ***
CreateAsteroid(6,x,y)
CreateAsteroid(6,x,y)
CreateAsteroid(6,x,y)
rem *** Destroy the original asteroid ***
DestroyAsteroid(id)
endfunction

```

Activity 18.19

The complete code for *Asteroids*:

```

rem *** Asteroid Game ***
rem *** Constants ***
#constant shipID = 101
#constant leftButtonID = 102
#constant rightButtonID = 103
#constant fireButtonID = 104
#constant firstMissileID = 105
rem *** Record Structures ***
type MissileType
    xoffset as float
    yoffset as float
endtype

type AsteroidType
    size
    xoffset as float
    yoffset as float
endtype

rem *** Global variables ***
global highestAsteroidID = 0
global rotateSpeed# = 1.0
global lastDirection = 103
global numberOfMissiles = 0
global buttonUp = 1
global dim asteroids[4] as AsteroidType
global dim missiles[5] as MissileType

rem *** Main Game Logic ***
LoadResources()
PositionShip()
PositionControls()
CreateAsteroid(12,Random(0,100),Random(0,100))
do
    MoveAllAsteroids()
    ControlShip()
    MoveAllMissiles()
    HandleCollisions()
    Sync()
loop
end

rem *****
rem *** Level 1 Functions ***
rem *****

function LoadResources()
    rem *** Asteroid images ***
    LoadImage(1,"Asteroid.png")
    LoadImage(2,"Explode01.png")
    LoadImage(3,"Explode02.png")
    LoadImage(4,"Explode03.png")
    rem *** Ship and missile images ***
    LoadImage(5,"Arrow.png")
    LoadImage(6,"Missile.png")
    rem *** Button images ***
    LoadImage(7,"Left.png")
    LoadImage(8,"Right.png")
    LoadImage(9,"Fire.png")
    rem *** Sound files ***
    LoadSound(1,"Launch.wav")
    LoadSound(2,"Explode.wav")
endfunction

function PositionShip()
    rem *** Create, size and position spacecraft ***
    CreateSprite(shipID,5)

```

```

SetSpriteSize(shipID,5,-1)
SetSpritePosition(shipID,47.5,48)
rem *** Rotate ship to point upwards ***
SetSpriteAngle(shipID,-90)
endfunction

```

```

function PositionControls()
    rem *** Position Left button ***
    CreateSprite(leftButtonID,7)
    SetSpriteSize(leftButtonID,10,-1)
    SetSpritePosition(leftButtonID,1,87)
    SetSpriteDepth(leftButtonID,9)
    rem *** Position right button ***
    CreateSprite(rightButtonID,8)
    SetSpriteSize(rightButtonID,10,-1)
    SetSpritePosition(rightButtonID,1,93)
    SetSpriteDepth(rightButtonID,9)
    rem *** Position fire button ***
    CreateSprite(fireButtonID,9)
    SetSpriteSize(fireButtonID,10,-1)
    SetSpritePosition(fireButtonID,89,93)
    SetSpriteDepth(fireButtonID,9)
endfunction

```

```

function CreateAsteroid(sz,x,y)
    inc highestAsteroidID
    id = highestAsteroidID
    rem *** Create, size and position sprite ***
    CreateSprite(id,1)
    SetSpriteSize(id,sz,-1)
    SetSpritePosition(id,x,y)
    rem *** Transform into animated sprite ***
    SetSpriteAnimation(id,250,250,16)
    AddSpriteAnimationFrame(id,2)
    AddSpriteAnimationFrame(id,3)
    AddSpriteAnimationFrame(id,4)
    rem *** Record sprite's size and velocity ***
    asteroids[id].size = sz
    asteroids[id].xoffset = (Random(0,20)-10)/10.0
    asteroids[id].yoffset = (Random(0,20)-10)/10.0
    PlaySprite(id,20,1,1,16)
endfunction

```

```

function MoveAllAsteroids()
    for c = 1 to highestAsteroidID
        if GetSpriteExists(c) = 1
            MoveSingleAsteroid(c)
        endif
    next c
endfunction

```

```

function ControlShip()
    rem *** Check for button pressed down ***
    x = GetPointerX()
    y = GetPointerY()
    id = GetSpriteHit(x,y)
    rem *** If button is pressed ***
    if GetPointerState() = 1
        rem *** Handle button ***
        select id
            case leftButtonID: //Rotate ship left
                SetSpriteAngle(shipID,
                    ↵GetSpriteAngle(shipID)-
                    ↵rotatespeed#)
            endcase
            case rightButtonID: //Rotate ship right
                SetSpriteAngle(shipID,
                    ↵GetSpriteAngle(shipID)+
                    ↵rotatespeed#)
            endcase
            case fireButtonID: //Handle missile firing
                rem *** If less than 5 missiles on
                ↵screen ***
                rem *** and fire button has just been
                ↵pressed ***
                if numberOfMissiles<5 and buttonup=1
                    rem *** Create a new missile ***
                    CreateMissile()
                    rem *** Record fire button as
                    ↵being pressed ***
                    buttonup = 0
                endif
            endcase
        endselect
    else
        rem *** If no buttons are pressed ***
        rem *** record fire button as up ***
        buttonup = 1
    endif
endfunction

```

```

function MoveAllMissiles()
    for c = firstMissileID to firstMissileID + 4
        if GetSpriteExists(c) = 1
            MoveSingleMissile(c)
        endif
    next c
endfunction

function HandleCollisions()
    rem *** FOR each missile ID DO ***
    for m = firstMissileID to firstMissileID + 4
        rem *** If the missile exists ***
        if GetSpriteExists(m) = 1
            rem *** FOR each asteroid ID DO ***
            for a = 1 to highestAsteroidID
                rem *** If asteroid exists ***
                if GetSpriteExists(a)=1
                    rem *** If misslie and asteroid
                    ↳have collided ***
                    if GetSpriteCollision(m,a) = 1
                        rem *** Delete missile and
                        ↳reduce missile count ***
                        DeleteSprite(m)
                        dec numberOfMissiles
                        rem *** If its a large
                        ↳asteroid split the asteroid
                        ↳***
                        if asteroids[a].size = 12
                            SplitAsteroid(a)
                        else
                            rem *** if its a small
                            ↳asteroid, destroy it
                            DestroyAsteroid(a)
                        endif
                        exit
                    endif
                endif
            next a
        endif
    next m
endfunction

rem *****
rem ***      Level 2 Functions      ***
rem *****

function MoveSingleAsteroid(id)
    rem *** Play animation ***
    rem *** Move sprite ***
    x# = GetSpriteX(id) + asteroids[id].xoffset
    if x# > 120
        x# = 0
    elseif x# < -20
        x# = 100
    endif
    y# = GetSpriteY(id) + asteroids[id].yoffset
    if y# > 120
        y# = 0
    elseif y# < -20
        y# = 100
    endif
    SetSpritePosition(id,x#,y#)
endfunction

function CreateMissile()
    rem *** Check pre-condition ***
    rem *** Exit if 5 missiles already exist ***
    if numberOfMissiles >= 5
        exitfunction
    endif
    rem *** Increment missile count ***
    inc numberOfMissiles
    rem For each missile ID available ***
    for c = firstMissileID to firstMissileID + 4
        rem *** If no missile of that ID ***
        if GetSpriteExists(c) = 0
            rem *** Create the missile under ship
            ↳***
            CreateSprite(c,6)
            SetSpriteDepth(c,11)
            SetSpriteSize(c,2,-1)
            SetSpritePositionByOffset(c,
            ↳GetSpriteXByOffset(shipID),
            ↳GetSpriteYByOffset(shipID))
            rem *** Rotate it to same angle as ship
            ↳***
            angle# = GetSpriteAngle(shipID)
            SetSpriteAngle(c,angle#)
            rem *** Set missiles velocity ***
            missiles[c-firstMissileID].xoffset =
            ↳cos(angle#)*2
            missiles[c-firstMissileID].yoffset =
            ↳sin(angle#)*2
            rem *** Play the launch sound ***
            PlaySound(1)
            rem *** Exit the FOR loop ***
            exit
        endif
    next c
endfunction

function MoveSingleMissile(id)
    rem *** If missile doesn't exist ***
    rem *** exit function ***
    if GetSpriteExists(id) = 0
        exitfunction
    endif
    rem *** Move the missile ***
    SetSpritePosition(id,GetSpriteX(id)+
    ↳missiles[id-firstMissileID].xoffset,
    ↳GetSpriteY(id)+missiles[id-firstMissileID].
    ↳yoffset)
    rem *** If missile leaves the screen ***
    rem *** delete it and decrease missile count ***
    if GetSpriteX(id) < 0 or GetSpriteX(id)> 105 or
    ↳GetSpriteY(id) < 0 or GetSpriteY(id) > 105
        DeleteSprite(id)
        dec numberOfMissiles
    endif
endfunction

function SplitAsteroid(id)
    rem *** If sprite doesn't exist, exit ***
    if GetSpriteExists(id) = 0
        exitfunction
    endif
    rem *** Get the asteroid's current position ***
    x = GetSpriteX(id)
    y = GetSpriteY(id)
    rem *** Create three new asteroids ***
    CreateAsteroid(6,x,y)
    CreateAsteroid(6,x,y)
    CreateAsteroid(6,x,y)
    rem *** Destroy the original asteroid ***
    DestroyAsteroid(id)
endfunction

function DestroyAsteroid(id)
    rem *** Play the explosion frames ***
    PlaySprite(id,10,1,17,19)
    Sync()
    rem *** Play explosion noise ***
    PlaySound(2)
    rem *** Delete sprite ***
    DeleteSprite(id)
endfunction

```


Screen Handling

In this Chapter:

- ☐ Accessing Screen Attributes
- ☐ Controlling Screen Orientation
- ☐ Screen and World Coordinates
- ☐ Zooming
- ☐ Scrolling
- ☐ Clipping
- ☐ Using Touch Statements
- ☐ Understanding Sync()
- ☐ Accessing the Backbuffer

Screen Handling

Introduction

Since an app is likely to run on various platforms, one of the things we need to take into account is the screen on which it is displayed.

In this section we describe the commands used to access information about the screen being used and how these commands affect the relationship between screen space and world space.

Screen-Related Statements

GetDeviceHeight()

We can discover the height of the current device's screen using the `GetDeviceHeight()` statement (see FIG-19.1).

FIG-19.1

`GetDeviceHeight()`

integer `GetDeviceHeight()` ()

The value returned is the actual resolution of the screen in pixels and normally assumes portrait orientation. The value will not change if you rotate the device.

GetDeviceWidth()

The width of the current device's display can be determined using `GetDeviceWidth()` (see FIG-19.2).

FIG-19.2

`GetDeviceWidth()`

integer `GetDeviceWidth()` ()

The value returned is the actual screen resolution in pixels and measures the width of the screen when used in portrait mode.

GetVirtualHeight() and GetVirtualWidth()

The `SetVirtualResolution()` statement was covered in Chapter 2.

If you have set your display to use virtual pixels (rather than the default percentage system), then you can retrieve the settings you specified in `SetVirtualResolution()` using `GetVirtualHeight()` and `GetVirtualWidth()` whose formats are shown in FIG-19.3.

FIG-19.3

`GetVirtualHeight()`

`GetVirtualWidth()`

integer `GetVirtualHeight()` ()

integer `GetVirtualWidth()` ()

Should you call these commands when using the default percentage screen setup, the return values for each statement will be 100.

GetDisplayAspect()

The width-to-height aspect ratio of the device's screen can be found using the `GetDisplayAspect()` statement (see FIG-19.4).

FIG-19.4

`GetDisplayAspect()`

float `GetDisplayAspect()` ()

GetOrientation()

If the device running your code is capable of detecting its own orientation, then you can use the `GetOrientation()` statement to determine if it is in landscape or portrait mode. The statement's format is shown in FIG-19.5.

FIG-19.5

integer `GetOrientation` ()

`GetOrientation()`

The statement returns one of four possible values. These are:

- 1 portrait mode
- 2 portrait mode - 180° rotation
- 3 landscape mode - 90° rotation counterclockwise
- 4 landscape mode - 90° rotation clockwise

Activity 19.1

Start a new project called *Orientation*.

Enter the following code in *main.agc*:

```
dim orientation[4] as string = ["", "Portrait",  
    "Inverted portrait", "Landscape right", "Landscape left"]  
CreateText(1, "")  
do  
    SetTextString(1, orientation[GetOrientation()])  
    Sync()  
loop
```

Transfer the code to your smart device and try rotating it to each of the various options and observing the results.

Save your project.

SetOrientationAllowed()

FIG-19.6

`SetOrientationAllowed()`

You have the option to disable a change of screen layout when a device's orientation shifts. Using `SetOrientationAllowed()` permits you to switch on or off automatic layout changes. The statement's format is shown in FIG-19.6.

`SetOrientationAllowed` (iport , ivport , iland , ivland)

where:

- | | |
|---------------|--|
| iport | is an integer value (0 or 1). Set this to 1 if standard portrait orientation is allowed. |
| ivport | is an integer value (0 or 1). Set this to 1 if inverted portrait orientation is allowed. |
| iland | is an integer value (0 or 1). Set this to 1 if standard landscape orientation (90° counterclockwise) is allowed. |
| ivland | is an integer value (0 or 1). Set this to 1 if inverted landscape orientation (90° clockwise) is allowed. |

A zero setting for any parameter means that particular orientation is not allowed. By default, all orientations are allowed.

Activity 19.2

Modify *Orientation* so that only standard portrait and standard landscape are allowed. See what effect this has when you run the app on your portable device.

SetTransitionMode()

If your app does allow a change of orientation, how that orientation is handled can be set using `SetTransitionMode()`. This statement allows you to specify either an instant change in orientation or a gradual one. Using the gradual option the screen image can be seen rotating through several stages to its new position. The format for `SetTransitionMode()` is shown in FIG-19. 7.

FIG-19.7

SetTransitionMode()

`SetTransitionMode ((imode))`

where:

imode is an integer value (0 or 1). 0: instant transition; 1: gradual transition.

The program in FIG-19.8 shows the effects of changing transition modes.

FIG-19.8

Using Transition Modes

```
rem *** Demonstrate Orientation Transition ***

rem *** Set screen aspect ***
SetDisplayAspect(768/1024.0)
rem *** Set transition effect ***
SetTransitionMode(1)
rem *** Create sprite ***
LoadImage(1,"Spacecraft.png")
CreateSprite(1,1)
SetSpriteSize(1,-1,50)
SetSpritePositionByOffset(1,50,50)
do
    Sync()
loop
```

Activity 19.3

Start a new project called *Transitions* and implement the code given in FIG-19.8 (also copy *AGKDownloads/Chapter19/Spacecraft.png* to the project's *media* folder.

Run the program on your mobile device and check out the effect as you move between landscape and portrait mode.

Change the parameter for `SetTransitionMode()` to zero and see how the effect produced changes.

Save your project.

SetResolutionMode()

When using a high resolution device such as your PC or tablet, you have the option to set the graphics of your app to high resolution or low resolution using the `SetResolutionMode()` statement (see FIG-19.9).

FIG-19.9

```
SetResolutionMode ( imode )
```

SetResolutionMode()

where:

imode is an integer value (0 or 1). Set to 1 for high resolution; 0 for low resolution.

Obviously, selecting high resolution will result in sharper images, but low resolution may give faster execution times.

UpdateDeviceSize()

If you are using virtual pixels, then width and height values can be changed (you may need to do this if the device is rotated) using the `UpdateDeviceSize()` statement (see FIG-19.10).

FIG-19.10

```
UpdateDeviceSize ( iwidth , iheight )
```

UpdateDeviceSize()

where

iwidth is an integer value giving the new width of the screen.

iheight is an integer value giving the new height of the screen.

Zooming and Scrolling

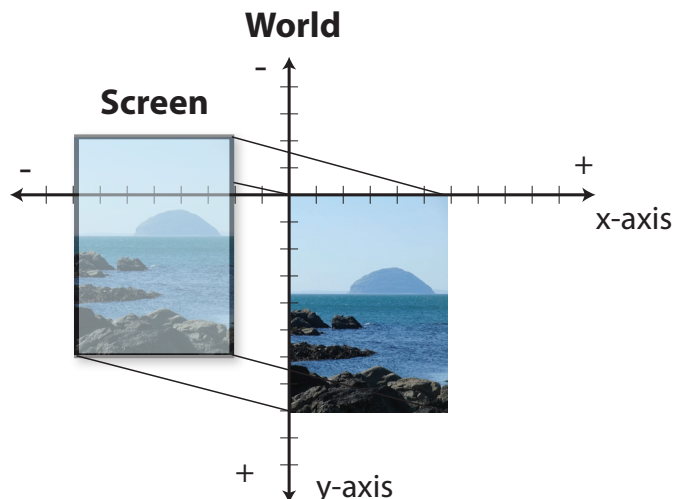
In order to understand zooming and scrolling, we need to know a few things about the virtual world created by AGK.

Although your device's screen (or app window on a PC) represents a 100% by 100% area, this is only part of the "world" in which visual objects can be placed. FIG-19.11 shows the relationship between the screen (or viewport) and the world.

FIG-19.11

World and Screen Coordinates

NOTE: markings on the axes are not to scale.



In this default setup, position (0,0) on the screen matches position (0,0) in the “world”. In fact all points between (0,0) and (100,100) match in both the screen and world setups.

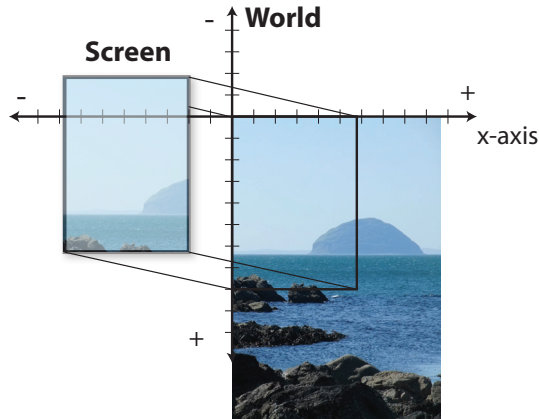
When we size an image to be larger than the screen with a line such as

```
SetSpriteSize(1,200,-1)
```

then part of the image lies in an area of the world not visible within the screen’s viewport (see FIG-19.12).

FIG-19.12

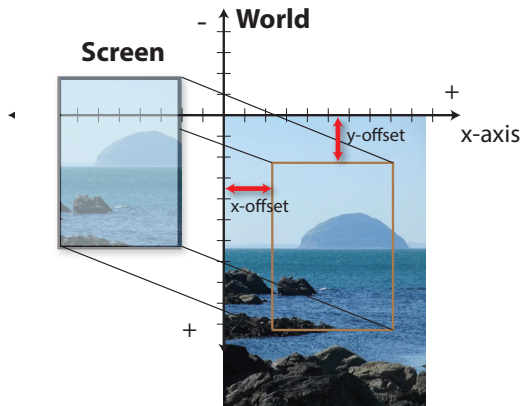
Resizing an Image



When a sprite is larger than the area of the screen, or when other visual elements have been placed outside the part of the “world” seen on the screen, then we need to use scrolling to see those off-screen elements (just like scrolling on a web page or text document). The idea is shown in FIG-19.13.

FIG-19.13

Scrolling



When we employ scrolling, the screen coordinates no longer match those of the world. For example, if we have scrolled 20 units in the x direction and 25 in the y direction (creating a similar situation to that in FIG-19.13 above), screen position (0,0) shows what is at position (20,25) in the world area.

When we adjust what appears on the screen by zooming, the image on the screen is a magnification of what’s been placed in the “world”. For example, if the image is sized using the line

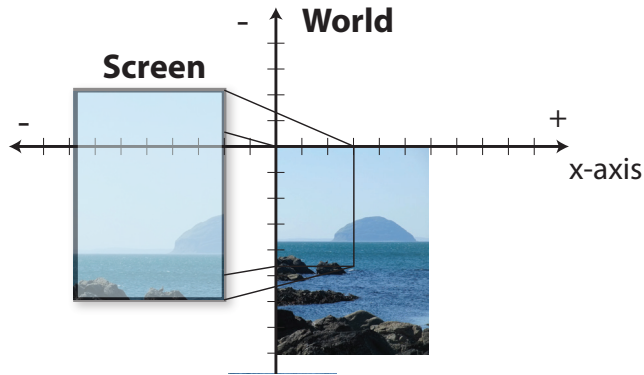
```
SetSpriteSize(1,100,-1)
```

and we zoom in on the screen so that we have a magnification factor of 2, then we

get the setup shown in FIG-19.14.

FIG-19.14

Using Zoom

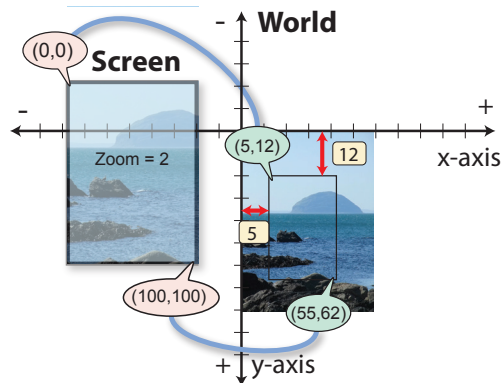


Like scrolling, this has affected how positions on the screen relate to (or map to) positions in the “world”. For example, in screen coordinates, the bottom-right corner is, as ever, (100,100) but this now corresponds to position (50,50) in world coordinates. If the magnification factor changed to 4, then the bottom-right of the screen would show what is at position (25,25) in the world.

If we use scrolling and zooming together mapping screen coordinates to world ones becomes even more difficult. If we assume we have offset the viewport by 12 units in the x direction and 5 units in the y direction and followed this up by using a magnification factor of 2, the resulting relationship between screen coordinates and world coordinates are shown in FIG-19.15.

FIG-19.15

Mapping Screen To World



When using scrolling and zooming, the following formulae are used to convert screen coordinates to world coordinates and vice versa:

$$x_{\text{world}} = x_{\text{screen}} / \text{magnification} + \text{xoffset} \quad x_{\text{screen}} = (x_{\text{world}} - \text{xoffset}) * \text{magnification}$$

$$y_{\text{world}} = y_{\text{screen}} / \text{magnification} + \text{yoffset} \quad y_{\text{screen}} = (y_{\text{world}} - \text{yoffset}) * \text{magnification}$$

Activity 19.4

If the screen is using a magnification of 2.5, and is offset in the x and y directions by 7 and -4 respectively, which position in the world maps to screen position (50,50)?

ScreenToWorldX() and ScreenToWorldY()

Although we have already discussed how to convert screen coordinates to world coordinates manually, there are two statements we can use to do the job for us. The `ScreenToWorldX()` and `ScreenToWorldY()` statements' formats are shown in FIG-19.16.

FIG-19.16

`ScreenToWorldX()` float `ScreenToWorldX` (`x`)

`ScreenToWorldY()` float `ScreenToWorldY` (`y`)

where

`x,y` are a pair of real values giving the screen coordinates to be converted to world coordinates.

WorldToScreenX() and WorldToScreenY()

Converting in the other direction, from world coordinates to screen ones, is achieved using the `WorldToScreenX()` and `WorldToScreenY()` statements (see FIG-19.17).

FIG-19.17

`WorldToScreenX()` float `WorldToScreenX` ()

`WorldToScreenY()` float `WorldToScreenY` ()

where

`x,y` are a pair of real value giving the world coordinates to be converted to screen coordinates.

SetViewZoom()

To create a zoom effect on the screen, we can use the `SetViewZoom()` statement (see FIG-19.18).

FIG-19.18

`SetViewZoom()` `SetViewZoom` (`fmag`)

where:

`fmag` is a real number giving the magnification factor to be used. A value of 0.5 would halve the size of the images on the screen; 1.0 would set them to normal size; 2.0 would double the image sizes.

The program in FIG-19.19 displays an image using three different zoom settings (1, 0.5 and 2) with a two second delay between each change.

FIG-19.19

Using Zoom()

```
rem *** Using Zoom ***

rem *** Load image ***
LoadImage(1,"AilsaCraig.jpg")

rem *** Display for 2 seconds ***
CreateSprite(1,1)
SetSpriteSize(1,100,-1)
Sync()
Sleep(2000)
```



FIG-19.19

(continued)

Using Zoom()

```
rem *** Zoom out ***
SetViewZoom(0.5)
Sync()
Sleep(2000)
rem *** Zoom in ***
SetViewZoom(2)
Sync()
do
loop
```

Activity 19.5

Start a new project called *Zooming* and implement the code given in FIG-19.19 copying *AilsaCraig.jpg* from *AGKDownloads/Chapter19* to the project's *media* folder. (Use screen size 768 x 1024.) Test and save your project.

SetViewZoomMode()

When you use zoom, the default setup is for the top-left corner of the screen to remain fixed. Should you want to change this so that it is the centre of the screen that remains fixed, you can do this by using the `SetViewZoomMode()` statement (see FIG-19.20).

FIG-19.20

SetViewZoomMode()

`SetViewZoomMode (imode)`

where:

imode is an integer value (0: top-left fixed, or 1: centre fixed).

Activity 19.6

Modify *Zooming* so that the centre of the image remains fixed when the screen magnification changes.

FIG-19.21 contains a program which allows the user to control the zoom in and zoom out options using two buttons.

FIG-19.21

User-Controlled Zoom()

```
rem *** Zooming ***

rem *** Load images ***
LoadImage(1,"AilsaCraig.jpg")
LoadImage(2,"In.png")
LoadImage(3,"Out.png")
rem *** Create main image ***
CreateSprite(1,1)
SetSpriteSize(1,100,-1)

rem *** Create zoom in and zoom out buttons ***
CreateSprite(2,2)
SetSpriteSize(2,12,-1)
SetSpritePosition(2,85,92)
CreateSprite(3,3)
SetSpriteSize(3,12,-1)
SetSpritePosition(3,2,92)
```



FIG-19.21

(continued)

User-Controlled Zoom()

```

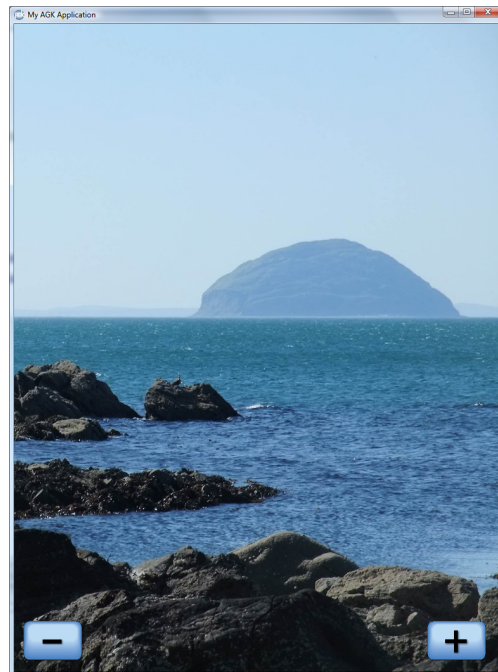
rem *** Image Title ***
CreateText(1,"Ailsa Craig, Scotland")
rem *** Zoom using buttons ***
zoom# = 1
do
    if GetPointerState() = 1
        if GetSpriteHit(GetPointerX(),GetPointerY()) = 2
            zoom# = zoom# +0.1
        elseif GetSpriteHit(GetPointerX(),GetPointerY())=3
            zoom# = zoom# -0.1
        endif
    endif
    SetViewZoom(zoom#)
    Sync()
    Sleep(100)
loop

```

The initial setup created by this program is shown in FIG-19.22.

FIG-19.22

Program Screen Shot

**Activity 19.7**

Start a new project called *Zooming2* and implement the code given in FIG-19.21. Copy the file *In.png*, *Out.png* and *AilsaCraig.png* from *AGKDownloads/Chapter19* to the *media* folder.

What problems arise when you test the program?

► Actually, if you press in the parts of the screen where the buttons had been, the zoom feature will still operate even though the buttons are no longer within the screen area.

As you can see from the results of Activity 19.7, zooming changes the size and position of all the images on the screen - even the text and zoom buttons. Worse, all the buttons move off-screen as we zoom making it impossible to press them.

FixSpriteToScreen()

If we want sprites such as the zoom buttons of the last project to stay in a fixed position, unaffected by the zooming, we can make use of the `FixSpriteToScreen()` statement (see FIG-19.23).

FIG-19.23

`FixSpriteToScreen()`

`FixSpriteToScreen (id , ifix)`

where:

- id** is an integer value giving the ID of the sprite.
- ifix** is an integer value (0 or 1) which determines if the sprite moves with changes to the screen magnification (0) or remains unchanged (1).

FixTextToScreen()

What we can do for sprites, we can also do for text objects. To fix a text object to its position on the screen we use `FixTextToScreen()` (see FIG-19.24).

FIG-19.24

`FixTextToScreen()`

`FixTextToScreen (id , ifix)`

where:

- id** is an integer value giving the ID of the text object.
- ifix** is an integer value (0 or 1) which determines if the text moves with changes to the screen magnification (0) or remains unchanged (1).

Activity 19.8

Modify *Zooming2* so that the two zoom buttons and image title remain fixed in position when the image is zoomed. Test your project. What problems arise this time?

Save your project.

FixEditBoxToScreen()

A final element that can be fixed to a screen position and hence remain unaffected by scrolling and zooming, is the edit box. To fix an edit box to its screen position, use `FixEditBoxToScreen()` (see FIG-19.25).

FIG-19.25

`FixEditBoxToScreen()`

`FixEditBoxToScreen (id , ifix)`

where:

- id** is an integer value giving the ID of the existing edit box.
- ifix** is an integer value (0 or 1) which determines if the edit box moves with zoom and scrolling changes (0) or remains in a fixed position (1).

Fixed Elements and World Coordinates

If an element such as a sprite or text resource stay fixed on the screen, it means that it must be shifting its position in the underlying world coordinate system when zooming or scrolling is used.

Many of the commands we have covered in the previous chapters have required *x* and *y* coordinates for positioning sprites or detecting hits. All of these statements require the *x* and *y* coordinates specified in the parameter list to be given in world coordinates. Functions that return *x* and *y* coordinates also generally give their results in world coordinates.

Of course, if the app is not using zooming or scrolling, then screen and world coordinates are the same and calling these functions doesn't present a problem. However, if the screen and world coordinates do not match, you always need to call `ScreenToWorldX()` and `ScreenToWorldY()` before passing parameters to any function requiring world coordinates.

Functions that require world coordinates include:

<code>GetSpriteHit()</code>	<code>GetSpriteHitTest()</code>
<code>SetSpritePosition()</code>	<code>SetSpritePositionByOffset()</code>
<code>SetSpriteX()</code>	<code>SetSpriteY()</code>

Functions that return screen coordinates include:

<code>GetPointerX()</code>	<code>GetPointerY()</code>
----------------------------	----------------------------

Now we arrive at the cause of our problem in the last Activity. When we press on the screen (or click with the mouse) we are given screen coordinates, but to check if a sprite has been hit (which we do with the buttons), we must supply world coordinates.

We can solve the problem in the last Activity by converting the screen touches to world coordinates with code such as:

```
ScreenToWorldX(GetPointerX())
```

Activity 19.9

Modify *Zooming2* so that the values returned by `GetPointer` calls are converted to world coordinates. How does this affect the program? Save your project.

GetViewZoom()

When using the zoom option, we can find the current magnification factor using the `GetViewZoom()` statement (see FIG-19.26).

FIG-19.26

GetViewZoom()

float `GetViewZoom()` ()

The function returns the current magnification factor. A value of 1.0 indicates that no magnification is being used.

Activity 19.10

Modify *Zooming2* so that the magnification factor is displayed in the top-left of the screen in place of the image title. Test and save your project.

SetViewOffset()

The trouble with using the zoom command is that, having zoomed in, there is no way of seeing the part of the image that is off-screen.

We can solve this problem by scrolling. Scrolling in AGK is achieved by using the `SetViewOffset()` statement which allows you to shift the viewport relative to the “world” coordinates (see FIG-19.27)

FIG-19.27

`SetViewOffset ((xoff , yoff)`

`SetViewOffset()`

where

xoff is a real value giving the offset in the *x* direction. This figure will be given as a percentage or in virtual pixels as appropriate.

yoff is a real number giving the offset in the *y* direction.

The program in FIG-19.28 is a variation on the previous program. It adds left and right scroll buttons to allow the user to scroll over the enlarged image.

FIG-19.28

Using Zoom and Scrolling

```
rem *** Zooming and Scrolling ***

rem *** Load images ***
LoadImage(1,"AilsaCraig.jpg")
LoadImage(2,"In.png")
LoadImage(3,"Out.png")
LoadImage(4,"Left.png")
LoadImage(5,"Right.png")

rem *** Create main screen image ***
CreateSprite(1,1)
SetSpriteSize(1,100,-1)

rem *** Create zoom in and zoom out buttons ***
CreateSprite(2,2)
SetSpriteSize(2,12,-1)
SetSpritePosition(2,85,92)
CreateSprite(3,3)
SetSpriteSize(3,12,-1)
SetSpritePosition(3,2,92)

rem *** Create scroll buttons ***
CreateSprite(4,4)
SetSpriteSize(4,10,-1)
SetSpritePosition(4,40,92)
CreateSprite(5,5)
SetSpriteSize(5,10,-1)
SetSpritePosition(5,50,92)

rem *** Fix buttons to screen ***
FixSpriteToScreen(2,1)
FixSpriteToScreen(3,1)
FixSpriteToScreen(4,1)
FixSpriteToScreen(5,1)

rem *** Zoom/Scroll using buttons ***
zoom# = 1
offsetX# = 0
```



FIG-19.28

(continued)

Using Zoom and Scrolling

```

do
    if GetPointerState() = 1
        select GetSpriteHit(ScreenToWorldX(GetPointerX()),
                           ScreenToWorldY(GetPointerY()))
            case 2: // Zoom In
                zoom# = zoom# + 0.1
            endcase
            case 3: // Zoom Out
                zoom# = zoom# - 0.1
            endcase
            case 4: // Scroll Left
                offsetX# = offsetX# - 1
            endcase
            case 5: // Scroll Right
                offsetX# = offsetX# + 1
            endcase
        endselect
    endif
    rem *** Set zoom factor ***
    SetViewZoom(zoom#)
    rem *** Set offset factor ***
    SetViewOffset(offsetX#, 0)
    Sync()
    Sleep(100)
loop

```

Use *Up.png* and *Down.png* images as the vertical scrolling buttons.

Activity 19.11

Start a new project called *ZoomScroll* and implement the code given in FIG-19.28. Copy all of the required files to the *media* folder.

Modify the program to include vertical as well as horizontal scrolling. Test and save your project.

GetViewOffsetX() and GetViewOffsetY()

We can discover the amount of viewport offset using the `GetViewOffsetX()` and `GetViewOffsetY()` statements (see FIG-19.29).

FIG-19.29

GetViewOffsetX()

GetViewOffsetY()

float `GetViewOffsetX()` ()float `GetViewOffsetY()` ()**Activity 19.12**

Modify *ZoomScroll* so that the *x* and *y* offset values are displayed at the top-left of the screen. Test and save your project.

Activity 19.13

Modify *ZoomScroll* so that the world coordinates of the mouse pointer are displayed instead of the *x* and *y* offsets.

Check out how the world coordinates initially match those of the screen and how zooming and offsetting modifies this relationship. Save your project.

Mouse Scrolling

The scroll buttons in the last project are a rather clumsy way of achieving our requirements. On a traditional computer screen, we could achieve scrolling by simply dragging the mouse. To use the mouse to scroll an image, we must detect the initial press of the mouse. When this happens, we then need to record the position of the mouse pointer. Now, while the mouse button remains pressed, we compare its new position with the starting position. The difference between these two values will indicate the amount of scrolling required. When the mouse button is released, scrolling terminates. An initial attempt at this approach is given in FIG-19.30.

FIG-19.30

Scrolling Using the
Mouse

```
rem *** Zooming and Scrolling ***

rem *** Load images ***
LoadImage(1,"AilsaCraig.jpg")
LoadImage(2,"In.png")
LoadImage(3,"Out.png")

rem *** Create main image ***
CreateSprite(1,1)
SetSpriteSize(1,100,-1)

rem *** Create zoom in and zoom out buttons ***
CreateSprite(2,2)
SetSpriteSize(2,12,-1)
SetSpritePosition(2,85,92)
CreateSprite(3,3)
SetSpriteSize(3,12,-1)
SetSpritePosition(3,2,92)

rem *** Fix buttons to screen ***
FixSpriteToScreen(2,1)
FixSpriteToScreen(3,1)

rem *** Zoom using buttons ***
zoom# = 1
do
    rem *** Check if mouse button just pressed ***
    pressed = GetPointerPressed()
    rem *** IF mouse button down ***
    if GetPointerState() = 1
        rem *** Operation depends on sprite hit ***
        select GetSpriteHit(ScreenToWorldX(GetPointerX()),
            ↵ScreenToWorldY(GetPointerY()))
            case 1: // Main image sprite
                rem *** IF mouse just pressed, save start
                ↵position ***
                if pressed = 1
                    x# = GetPointerX()
                    y# = GetPointerY()
                else
                    rem *** else record latest position ***
                    newx# = GetPointerX()
                    newy# = GetPointerY()
                endif
            endcase
            case 2: // Zoom In sprite
                zoom# = zoom# +0.1
            endcase
        endselect
    enddo
```



FIG-19.30

(continued)

Scrolling Using the
Mouse

```

        case 3: // Zoom Out sprite
            zoom# = zoom# -0.1
        endcase
    endselect
endif
rem *** Modify zoom level ***
SetViewZoom(zoom#)
rem *** If mouse being dragged, modify image (start position
↳ - current)***
if pressed = 0
    SetViewOffset(x#-newx#,y#-newy#)
endif
Sync()
loop

```

Activity 19.14

Modify *ZoomScroll* to match the code given in FIG-19.30.

Test your code. What happens when you attempt a second scroll? Save your project.

The problem with the existing code is that it does not take into account any scrolling that has taken place previously. To solve this problem, we need to record the existing scrolling when the mouse button is initially pressed. This can be achieved with the lines:

```

xoffsetatstart# = GetViewOffsetX()
yoffsetatstart# = GetViewOffsetY()

```

We then need to use these two values when redrawing the image. This means that the line

```
SetViewOffset(x#-newx#,y#-newy#)
```

must be changed to

```

SetViewOffset(xoffsetatstart#+x#-newx#,yoffsetatstart#+
↳ y#-newy#)

```

Activity 19.15

Modify *ZoomScroll* using the code above to correct the problem with the scrolling. Test your program.

How is scrolling affected when you have previously zoomed in on the image? Save your project.

The last problem is to match the movement of the mouse when the screen is zoomed to the equivalent displacement in the world space.

For example, if the magnification factor is set to 2 and the mouse is dragged from one side of the screen to the other then, on the screen we have moved 100 units but, because of the magnification factor, this is equivalent of only 50 units in world space.

So we need to make one last change to how the offset is calculated:

```
SetViewOffset(xoffsetatstart#+(x#-newx#)/zoom#,yoffsetatstart#+
↳ (y#-newy#)/zoom#)
```

Activity 19.16

Modify *ZoomScroll* so that the scrolling works correctly when the image is zoomed. Test and save your project.

We'll look at a way of replacing the zoom buttons in the next section.

Touch Statements

Although we can use the various `GetPointer` functions to detect both screen touches and mouse movements, we can achieve a lot more by making use of specific screen touch commands if we know our app will be running on a touch screen device.

GetMultiTouchExists()

To check if a device supports multiple simultaneous touches, we can use the `GetMultiTouchExists()` statement (see FIG-19.31).

FIG-19.31

integer `GetMultiTouchExists` () ()

`GetMultiTouchExists()`

The function returns 1 if the device supports multi-touch, otherwise zero is returned.

GetRawTouchCount()

The number of simultaneous touches on a screen can be determined using the `GetRawTouchCount()` statement. AGK normally delays the detection of a touch until it has determined the nature of that touch. There are three categories: short, long and drag. The `GetRawTouchCount()` statement can return a count of identified touches only or of all touches including those whose nature has not yet been determined.

The format for the `GetRawTouchCount()` statement is shown in FIG-19.32.

FIG-19.32

integer `GetRawTouchCount` ((itype))

`GetRawTouchCount()`

where

itype is an integer value (0 or 1) used to determine the type of touches to be included in the count (0: identified touches only; 1: all touches).

The program in FIG-19.33 displays a count of all touches being made on the screen.

FIG-19.33

Counting Touches

```
rem *** Count Touches ***
do
    Print(Str(GetRawTouchCount(1)))
    Sync()
loop
```

Activity 19.17

Create a new project called *Touch01* and implement the code in FIG-19.33. Test the program on your touch device and check the count displayed.

Modify the program so that only recognised touches are displayed. How does this affect the result?

Save your project.

GetRawFirstTouchEvent()

Details of every touch event are recorded in a list. To access the first unhandled item in that list we need to use the `GetRawFirstTouchEvent()`. This function returns the position of the first unhandled item in the list. If the list is empty, zero is returned.

FIG-19.34

`GetRawFirstTouchEvent()`

integer `GetRawFirstTouchEvent` ((`itype`))

where

itype is an integer value used to determine the type of touches to be accessed in the list (0: identified touches only; 1: all touches).

GetRawNextTouchEvent()

Subsequent entries in the touch event list must be accessed using `GetRawNextTouchEvent()`. If the list is empty, zero is returned.

The `GetRawNextTouchEvent()` statement has the format shown in FIG-19.35.

FIG-19.35

`GetRawNextTouchEvent()`

integer `GetRawNextTouchEvent` (())

GetRawTouchType()

Once a touch event has been discovered using `GetRawFirstTouchEvent()` or `GetRawNextTouchEvent()`, various information about that touch can be retrieved. One such piece of information is the type of touch - short, long, drag, or unidentified (1, 2, 3 or 0).

The type of touch can be found using the `GetRawTouchType()` statement (see FIG-19.36).

FIG-19.36

`GetRawTouchType()`

integer `GetRawTouchType` ((`id`))

where

id is an integer value giving the position of the touch event in the event list.

The program in FIG-19.37 displays the type for every touch detected.

FIG-19.37

Displaying Touch Type

```

rem *** Touch Types ***
rem *** Create text ***
CreateText(1,"")
do
    rem *** Get first touch ***
    id = GetRawFirstTouchEvent(0)
    while id <> 0
        rem *** Display its type ***
        SetTextString(1,Str(GetRawTouchType(id)))
        rem *** Get next touch ***
        id = GetRawNextTouchEvent()
    endwhile
    Sync()
loop

```

Activity 19.18

Create a new project called *Touch02* and implement the code in FIG-19.37.

Test the program on your PC using the mouse.

Modify the program so that unrecognised touches are displayed. Save your project.

GetRawTouchTime()

We can discover the duration of a touch (in seconds) using the `GetRawTouchTime()` statement (see FIG-19.38).

FIG-19.38

GetRawTouchTime()

float `GetRawTouchTime` (`id`)

where

id is an integer value giving the position of the touch event in the event list.

Activity 19.19

Modify *Touch02* so that it displays the duration of each screen press instead of its type. Test the program on your PC using the mouse. Save your project.

GetRawTouchReleased()

The `GetRawTouchReleased()` function returns 1 the instant a touch/mouse button is released. If the touch is still in progress (as it may be for long and drag touches), the function returns zero. The statement's format is shown in FIG-19.39.

FIG-19.39GetRawTouch
Released()integer `GetRawTouchEventReleased` (`id`)

where

id is an integer value giving the position of the touch event in the event list.

GetRawTouchStartX() and GetRawTouchStartY()

You can find the screen coordinates of where a touch starts using the `GetRawTouchStartX()` and `GetRawTouchStartY()` statements (see FIG-19.40).

FIG-19.40

`GetRawTouchStartX()`

`GetRawTouchStartY()`

float `GetRawTouchStartX` (`id`)

float `GetRawTouchStartY` (`id`)

where

id is an integer value giving the position of the touch event in the event list.

GetRawTouchCurrentX() and GetRawTouchCurrentY()

To discover the current screen coordinates of a touch, use `GetRawTouchCurrentX()` and `GetRawTouchCurrentY()` (see FIG-19.41).

FIG-19.41

`GetRawTouchCurrentX()`

`GetRawTouchCurrentY()`

float `GetRawTouchCurrentX` (`id`)

float `GetRawTouchCurrentY` (`id`)

where

id is an integer value giving the position of the touch event in the event list.

GetRawTouchLastX() and GetRawTouchLastY()

To discover the final screen coordinates of a touch at the moment the user's finger is lifted from the screen (or the mouse button released), use `GetRawTouchLastX()` and `GetRawTouchLastY()` (see FIG-19.42).

FIG-19.42

`GetRawTouchLastX()`

`GetRawTouchLastY()`

float `GetRawTouchLastX` (`id`)

float `GetRawTouchLastY` (`id`)

where

id is an integer value giving the position of the touch event in the event list.

SetRawTouchValue()

If you wish, you may assign a value to a specific touch ID using `SetRawTouchValue()` (see FIG-19.43).

FIG-19.43

`SetRawTouchValue()`

`SetRawTouchValue` (`id` , `ival`)

where

id is an integer value giving the position of the touch event in the event list.

ival is the integer number to be associated with the specified touch ID.

GetRawTouchValue()

The value assigned to a touch can be retrieved using the `GetRawTouchValue()` statement (see FIG-19.44).

FIG-19.44

`integer GetRawTouchValue (id)`
where

id is an integer value giving the position of the touch event in the event list.

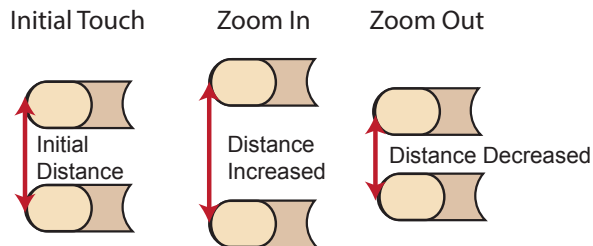
Using Touch to Zoom and Scroll

Earlier we used the mouse to scroll and zoom an image. In this next example, we'll use a single touch to scroll and a double touch to zoom. The scrolling technique is no different from that we used earlier: get the initial touch position and use the difference between that and the latest position to scroll.

To zoom, we will take the difference in the y coordinates of a double touch to decide the magnification factor. The initial distance between the two elements of the double touch will represent the current zoom factor and any change in that distance will adjust that factor (see FIG-19.45).

FIG-19.45

Calculating Zoom Values



The code for the program is given in FIG-19.46.

FIG-19.46

Touch Controlled Zoom and Scroll

```
rem *** Touch Zooming and Scrolling ***  
  
rem *** Set aspect ratio ***  
SetDisplayAspect(768/1024.0)  
  
rem *** Load image ***  
LoadImage(1,"AilsaCraig.jpg")  
  
rem *** Create Sprite ***  
CreateSprite(1,1)  
SetSpriteSize(1,100,-1)  
  
rem *** Zoom factor ***  
lastzoom# = 1  
zoom#=1  
do  
    touches = GetRawTouchCount(1)  
    select touches
```



FIG-19.46

(continued)

Touch Controlled Zoom
and Scroll

```

case 0:
    rem *** Record current zoom setting ***
    lastzoom# = zoom#
endcase
case 1: // Scroll
    rem *** Get current offsets ***
    xoffsetatstart# = GetViewOffsetX()
    yoffsetatstart# = GetViewOffsetY()
    rem *** Get coords at start of touch ***
    id = GetRawFirstTouchEvent(0)
    x# = GetRawTouchStartX(id)
    y# = GetRawTouchStartY(id)
    rem *** Read current coords and offset the image ***
    while GetRawTouchReleased(id) = 0
        currentx# = GetRawTouchCurrentX(id)
        currenty# = GetRawTouchCurrentY(id)
        SetViewOffset(xoffsetatstart# + (x# - currentx#) /
            ↳ lastzoom#, yoffsetatstart# + (y# - currenty#) / lastzoom#)
        Sync()
    endwhile
endcase
case 2: // Zoom
    rem *** Get start y coord for each touch ***
    id1 = GetRawFirstTouchEvent(1)
    id2 = GetRawNextTouchEvent()
    y1 = GetRawTouchStartY(id1)
    y2 = GetRawTouchStartY(id2)
    rem *** Calculate the distance between the two points
    ↳ ***
    dist# = Abs(y1 - y2)
    rem *** Get new distance appart, compare with original
    ↳ and adjust zoom accordingly ***
    while GetRawTouchReleased(id1) = 0 and
        ↳ GetRawTouchReleased(id2) = 0
        newdist# = Abs(GetRawTouchCurrentY(id1) -
            ↳ GetRawTouchCurrentY(id2))
        diff# = Abs(newdist# / dist#)
        zoom# = lastzoom# * diff#
        SetViewZoom(zoom#)
        Sync()
    endwhile
endcase
endselect
Sync()
loop

```

Activity 19.20

Start a new project called *ZoomScrollTouch* and implement the code given in FIG-19.46 (copy the necessary image into the *media* folder).

Test and save your project.

SetScissor()

Another, but quite different way of adjusting how the world elements are mapped onto the screen is achieved using the `SetScissor()` statement. This statement crops all elements outside a specified area of the screen. The `SetScissor()` statement has

the format shown in FIG-19.47.

FIG-19.47

SetScissor()

SetScissor ((x1 , y1 , x2 , y2))

where

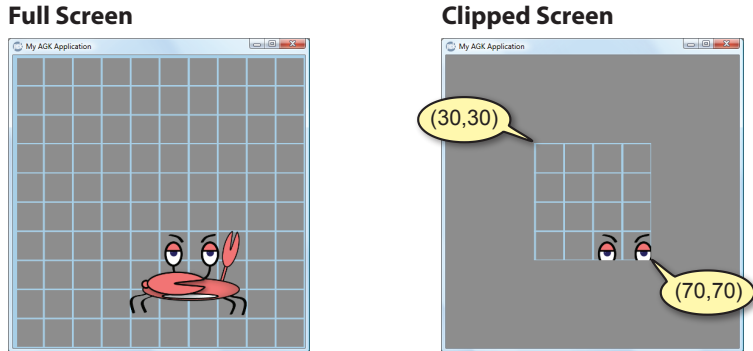
x1,y1 are a pair of real values giving the coordinates of the top-left corner of the clipping area.

x2,y2 are a pair of real values giving the coordinates of the bottom-right corner of the clipping area.

FIG-19.48 shows a screen display before and after clipping.

FIG-19.48

The Effects of Clipping



The program that produced the screen displays is shown in FIG-19.49. The code shows the normal screen for 3 seconds before clipping.

FIG-19.49

Using Clipping

```
rem *** Screen Clipping ***

rem *** Load images ***
LoadImage(1,"Grid.png")
LoadImage(2,"Crab.png")
rem *** Set screen background ***
SetClearColor(140,140,140)
Sync()
rem *** Create sprites ***
CreateSprite(1,1)
SetSpriteSize(1,100,100)
CreateSprite(2,2)
SetSpriteSize(2,40,-1)
SetSpritePosition(2,40,60)
Sync()
Sleep(3000)
rem *** Clip screen ***
SetScissor(30,30,70,70)
do
    Sync()
loop
```

Activity 19.21

Start a new project called *Clipping* and implement the code given in FIG-19.49 (copy the necessary images into the *media* folder). Test and save your project.

- Use `GetDeviceHeight()` and `GetDeviceWidth()` to determine the dimensions of the device running your app. Values given in pixels.
- When your app has been set up to use virtual pixels, use `GetVirtualHeight()` and `GetVirtualWidth()` to discover the current dimensions of the screen when using virtual pixels.
- Use `GetDisplayAspect()` to find the aspect ratio set for the screen in the last call to `SetDisplayAspect()`.
- Use `GetOrientation()` to detect the orientation of any device which contains an accelerometer.
- Use `SetOrientationAllowed()` to specify which orientations will automatically reorientate the screen display.
- Use `SetTransitionMode()` to specify how the display moves from one orientation to another.
- Use `SetResolutionMode()` to specify the resolution setting for screen graphics.
- Use `UpdateDeviceSize()` to change the virtual pixel dimensions of the screen.
- AGK makes use of two coordinates systems: screen coordinates and world coordinates.
- When positioning visual elements they are placed using world coordinates.
- Screen coordinates and world coordinates match exactly if no scrolling or zooming has occurred.
- Use `SetViewZoom()` to change screen magnification.
- Use `SetViewZoomMode()` to specify if the top-left or centre point is to remain fixed when zooming.
- Use `FixSpriteToScreen()` to fix the position of a sprite on the screen irrespective of zooming and scrolling.
- Use `FixTextToScreen()` to fix the position of text on the screen.
- Use `FixEditBoxToScreen()` to fix an edit box to screen space.
- Use `GetViewZoom()` to discover the current magnification factor being used.
- Offsetting the screen changes which part of world space appears on the screen.
- Use `SetViewOffset()` to offset the screen's view.
- Use `GetViewOffsetX()` and `GetViewOffsetY()` to determine the current screen offsets.
- Use `ScreenToWorldX()` and `ScreenToWorldY()` to convert screen coordinates to world coordinates.
- Use `WorldToScreenX()` and `WorldToScreenY()` to convert world coordinates to screen coordinates.
- Use `MultiTouchExists()` to determine if multiple simultaneous screen touches are detected.

- Use `GetRawTouchCount()` to detect the current number of simultaneous touches.
- Use `GetRawFirstTouchEvent()` to determine the ID of the first touch event listed.
- Use `GetRawNextTouchEvent()` to determine the ID of subsequent touches listed.
- Use `GetRawTouchType()` to determine the type of touch detected (short, long, drag or unidentified).
- Use `GetRawTouchTime()` to discover the current duration of a specific touch.
- Use `GetRawTouchReleased()` to detect the moment a specified touch is released.
- Use `GetRawTouchStartX()` and `GetRawTouchStartY()` to determine the start coordinates of a touch.
- Use `GetRawTouchCurrentX()` and `GetRawTouchCurrentY()` to determine the current coordinates of a touch.
- Use `GetRawTouchLastX()` and `GetRawTouchLastY()` to determine the last coordinates of a touch at the moment of release.
- Use `SetRawTouchValue()` to associate a value with a specified touch.
- Use `GetRawTouchValue()` to retrieve the value associated with a specific touch.
- Use `SetScissor()` to clip all visual elements outside a specified area of the screen.

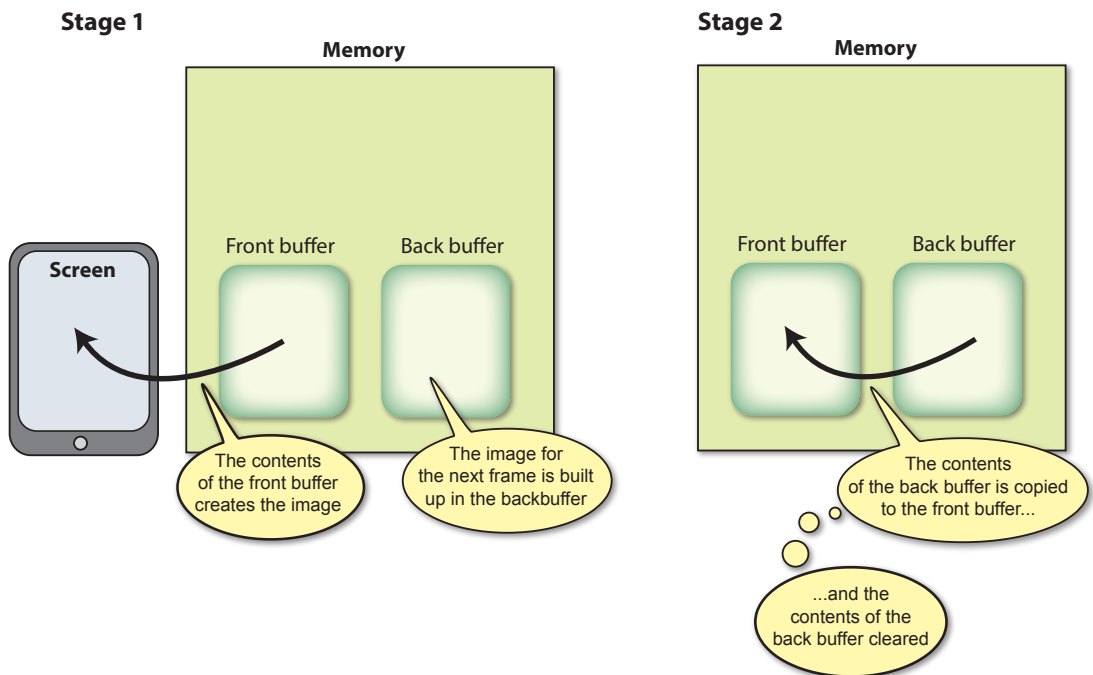
Secrets of Sync()

The contents of your screen are updated several times a second. Each update redraws the entire contents of the screen. Each redrawing is known as a **frame**.

To create a screen display, AGK reserves two areas of memory within your device. These areas of memory are known as **screen buffers**. The contents of one buffer are used to create the frame currently being displayed on the device's screen. This is known as the **screen buffer** or **front buffer**. At the same time, the contents of the second buffer (known as the **back buffer**) are being updated to contain the layout of the next frame.

FIG-19.50 shows how these buffers are used in the construction of a frame.

FIG-19.50 How the Screen Display is Produced



Notice that after the contents of the front buffer is displayed, the contents of the back buffer are transferred to the front buffer, ready to display the next frame.

The `Sync()` function, which updates all-screen related details, does, in fact, call three other routines which handle the various aspects of screen updating. Under normal circumstances, we are quite happy to call `Sync()` and let it handle the screen, but we have the option to replace `Sync()` and call the three routines ourselves. Doing so might, in certain circumstances, allow us to achieve a higher frame rate when we have complex calculations to perform between frames.

Update()

The `Update()` function is used to update the state of various objects on the screen. For example, it will calculate the layout of the latest frame to be displayed when

handling animated sprites and physics calculations when using the physics engine. The statement has the format shown in FIG-19.51.

FIG-19.51

Update()

Update (interval)

where

interval is a real number giving the time (in seconds) assumed to have passed since the last update. This time is important when you are using physics. If zero is given, then the time interval is assumed to be the time taken to build the previous frame (typically 0.017).

Render()

The second function called by `Sync()` is `Render()`. The `Render()` function actually creates the new screen image data in the back buffer. For example, it will make sure that the images of any sprites, text objects, edit boxes, etc have been placed in the back buffer. The format for this statement is given in FIG-19.52.

FIG-19.52

Render()

Render ()

Swap()

The final function of the trio is `Swap()`. This function swaps which buffer is being used as the front buffer and clears the contents of the second buffer (now the new back buffer). In addition, it also updates various time-related variables. The format for `Swap()` is given in FIG-19.53.

FIG-19.53

Swap()

Swap ()

Activity 19.22

In *MovingBall2* which you created in Chapter 17, remove the call to `Sync()` and replace it with the lines

```
Update (0)
Render ()
Swap ()
```

How does this affect the program?

ClearScreen()

The `ClearScreen()` function clears the contents of the back buffer and fills it with the current clear colour (as set using `SetClearColor()`). The statement has the format shown in FIG-19.54.

FIG-19.54

ClearScreen()

ClearScreen ()

However, remember that the statement actually clears the back buffer and not the front buffer, so it will not clear the screen you see in front of you. And when you call `Sync()` its call to `Render()` will repopulate the back buffer with the various visual elements. However, the lines

```
ClearScreen()
Swap()
```

will create a cleared screen until the next `Sync()` statement is executed.

DrawSprite()

Another statement which modifies the back buffer directly is `DrawSprite()` which draws the specified sprite onto the back buffer. The statement's format is shown in FIG-19.55.

FIG-19.55

`DrawSprite()`

```
DrawSprite ( id )
```

where

id is an integer value giving the ID of the sprite whose image is to be written to the back buffer.

The sprite is drawn with its existing attributes (size and rotation).

GetImage()

If you want to copy part of the screen image built up in the back buffer, this can be achieved using the `GetImage()` statement which will turn the captured area of the buffer into another image. The statement's format is shown in FIG-19.56.

FIG-19.56

`GetImage()`

Version 1

```
GetImage ( id , x , y , width , height )
```

Version 2

```
integer GetImage ( x , y , width , height )
```

where

id is an integer value giving the ID to be assigned to the new image.

x,y are real values giving the coordinates of the top-left corner of the area to be copied. Use percentage or virtual pixels as appropriate.

width is a real value giving the width of the area to be copied (in percentage or virtual pixels).

height is a real value giving the height of the area to be copied (in percentage or virtual pixels).

The function returns the ID assigned to the newly created image.

This can be a tricky statement to use the first time you come across it. If your program just makes use of the `Sync()` statement to handle the two screen buffers, then the back buffer will be empty so there's nothing in there to capture. The way round this is to replace the call to `Sync()` with your own calls to `Update()`, `Render()` and `Swap()` (functions called by `Sync()`). Then, after calling `Render()` but before calling `Swap()` you can use `GetImage()` to capture part of the back buffer.

Another method is to make direct use of the back buffer by using `ClearScreen()` and `DrawSprite()` before calling `GetImage()`.

For example, the line

```
id = GetImage(0,0,100,50)
```

would capture the top half of the back buffer's screen data and store that as an image.

The program in FIG-19.57 demonstrates the use of several of the previous statements by emptying the back buffer, drawing an image there, and then capturing part of that image. After a delay of one second the captured image is assigned to a sprite and displayed on the screen.

FIG-19.57

Accessing the Back
Buffer

```
rem *** Using the Back Buffer ***

rem *** Load image ***
LoadImage(1,"Grid.png")

rem *** Set screen background ***
SetClearColor(140,140,140)
Sync()

rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,100,100)

rem *** Clear the back buffer ***
ClearScreen()

rem *** Draw the grid in the back buffer ***
DrawSprite(1)

rem *** Capture a section of the back buffer ***
id = GetImage(30,30,40,40)

rem *** Delete the original sprite ***
DeleteSprite(1)

rem *** Clear the back buffer ***
ClearScreen()

rem *** Wait 1 second ***
CreateText(1,"Wait one second ...")
Sync()
Sleep(1000)

rem *** Delete the text ***
DeleteText(1)

rem *** Show the captured image on a sprite ***
CreateSprite(2,id)
SetSpriteSize(2,30,-1)

do
    Sync()
loop
```

Activity 19.23

Start a new project called *Backbuffer* and implement the code in FIG-19.57.

Copy *Grid.png* to the *media* folder.

Test and save your project.

Summary

- `Sync()` operates by calling the functions `Update()`, `Render()` and `Swap()`.
- The `Update()` operation calculates the new positions of physics-enabled objects and the current frame of any animated sprites.
- `Render()` builds the next screen image to appear on the screen in the back buffer.
- `Swap()` transfers the contents of the back buffer to the screen buffer and clears the back buffer's contents.
- Use `ClearScreen()` to clear the contents of the back buffer.
- Use `DrawSprite()` to place a sprite in the back buffer.
- Use `GetImage()` to capture an area of the back buffer and convert it to an image.

Solutions

Activity 19.1

No solution required.

Activity 19.2

Modified code for *Orientation*:

```
dim orientation[4] as string = ["",  
    ⚡"Portrait", "Inverted portrait",  
    ⚡"Landscape right", "Landscape left"]  
CreateText(1, "")  
rem *** Only standard portrait and landscape ***  
SetOrientationAllowed(1,0,1,0)  
do  
    SetTextString(1,orientation[GetOrientation()])  
    Sync()  
loop
```

Activity 19.3

Using the original code, the image rotates through several stages before settling in the new orientation.

With the second option the image jumps instantly from one orientation to the other.

Activity 19.4

(27,16) that is (50/2.5+7,50/2.5-4)

Activity 19.5

No solution required.

Activity 19.6

Modified code for *Zooming*:

```
rem *** Using Zoom ***  
  
rem *** Load image ***  
LoadImage(1,"AilsaCraig.jpg")  
rem *** Display for 2 seconds ***  
CreateSprite(1,1)  
SetSpriteSize(1,100,-1)  
Sync()  
Sleep(2000)  
rem *** Fix centre of image ***  
SetViewZoomMode(1)  
rem *** Zoom out ***  
SetViewZoom(0.5)  
Sync()  
Sleep(2000)  
rem *** Zoom in ***  
SetViewZoom(2)  
Sync()  
do  
loop
```

Activity 19.7

As the zoom increases, the buttons disappear off the screen.

Activity 19.8

Modified code for *Zooming2*:

```
rem *** Zooming ***  
  
rem *** Load images ***  
LoadImage(1,"AilsaCraig.jpg")  
LoadImage(2,"In.png")  
LoadImage(3,"Out.png")  
rem *** Create main image ***  
CreateSprite(1,1)  
SetSpriteSize(1,100,-1)
```

```
rem *** Create zoom in and zoom out buttons ***  
CreateSprite(2,2)  
SetSpriteSize(2,12,-1)  
SetSpritePosition(2,85,92)  
CreateSprite(3,3)  
SetSpriteSize(3,12,-1)  
SetSpritePosition(3,2,92)  
rem *** Fix buttons to screen ***  
FixSpriteToScreen(2,1)  
FixSpriteToScreen(3,1)  
rem *** Image Title ***  
CreateText(1,"Ailsa Craig, Scotland")  
rem *** Fix text to screen ***  
FixTextToScreen(1,1)  
rem *** Zoom using buttons ***  
zoom# = 1  
do  
    if GetPointerState() = 1  
        if GetSpriteHit(GetPointerX(),  
            ⚡GetPointerY()) = 2  
            zoom# = zoom# + 0.1  
        elseif GetSpriteHit(GetPointerX(),  
            ⚡GetPointerY()) = 3  
            zoom# = zoom# - 0.1  
        endif  
    endif  
    SetViewZoom(zoom#)  
    Sync()  
    Sleep(100)  
loop
```

The problem now is that the buttons no longer seem to respond to being pressed.

Activity 19.9

To convert screen touch coordinates to world coordinates, change the following lines

```
if GetSpriteHit(GetPointerX(),GetPointerY()) = 2  
    zoom# = zoom#+0.1  
elseif GetSpriteHit(GetPointerX(),GetPointerY()) = 3  
    zoom# = zoom# -0.1  
endif
```

to

```
if GetSpriteHit(ScreenToWorldX(GetPointerX()),  
    ⚡ScreenToWorldY(GetPointerY())) = 2  
    zoom# = zoom# + 0.1  
elseif GetSpriteHit(ScreenToWorldX(GetPointerX()),  
    ⚡ScreenToWorldY(GetPointerY())) = 3  
    zoom# = zoom# - 0.1  
endif
```

The zoom button should now respond correctly to being hit.

Activity 19.10

Modified code for *Zooming2*:

```
rem *** Zooming ***  
  
rem *** Load images ***  
LoadImage(1,"AilsaCraig.jpg")  
LoadImage(2,"In.png")  
LoadImage(3,"Out.png")  
rem *** Create main image ***  
CreateSprite(1,1)  
SetSpriteSize(1,100,-1)  
rem *** Create zoom in and zoom out buttons ***  
CreateSprite(2,2)  
SetSpriteSize(2,12,-1)  
SetSpritePosition(2,85,92)  
CreateSprite(3,3)  
SetSpriteSize(3,12,-1)  
SetSpritePosition(3,2,92)  
rem *** Fix buttons to screen ***  
FixSpriteToScreen(2,1)  
FixSpriteToScreen(3,1)  
rem *** Zoom Factor ***  
CreateText(1,"Zoom : 1.0")  
rem *** Fix text to screen ***  
FixTextToScreen(1,1)  
rem *** Zoom using buttons ***  
zoom# = 1  
do  
    if GetPointerState() = 1
```

```

    if GetSpriteHit(ScreenToWorldX(GetPointerX()),
        ↵ScreenToWorldY(GetPointerY())) = 2
        zoom# = zoom# + 0.1
    elseif GetSpriteHit(ScreenToWorldX(
        ↵GetPointerX()), ScreenToWorldY(GetPointerY()))
        ↵= 3
        zoom# = zoom# - 0.1
    endif
endif
SetViewZoom(zoom#)
rem *** Update text ***
SetTextString(1,"Zoom:"+Str(GetViewZoom()))
Sync()
Sleep(100)
loop

```

Activity 19.11

Modified code for *ZoomScroll*:

```

rem *** Zooming and Scrolling ***

rem *** Load images ***
LoadImage(1,"AilsaCraig.jpg")
LoadImage(2,"In.png")
LoadImage(3,"Out.png")
LoadImage(4,"Left.png")
LoadImage(5,"Right.png")
LoadImage(6,"Up.png")
LoadImage(7,"Down.png")
rem *** Create main image ***
CreateSprite(1,1)
SetSpriteSize(1,100,-1)
rem *** Create zoom in and zoom out buttons ***
CreateSprite(2,2)
SetSpriteSize(2,12,-1)
SetSpritePosition(2,85,92)
CreateSprite(3,3)
SetSpriteSize(3,12,-1)
SetSpritePosition(3,2,92)
rem *** Create scroll buttons ***
CreateSprite(4,4)
SetSpriteSize(4,10,-1)
SetSpritePosition(4,40,85)
CreateSprite(5,5)
SetSpriteSize(5,10,-1)
SetSpritePosition(5,50,85)
CreateSprite(6,6)
SetSpriteSize(6,10,-1)
SetSpritePosition(6,45,79)
CreateSprite(7,7)
SetSpriteSize(7,10,-1)
SetSpritePosition(7,45,91.5)
rem *** Fix buttons to screen ***
FixSpriteToScreen(2,1)
FixSpriteToScreen(3,1)
FixSpriteToScreen(4,1)
FixSpriteToScreen(5,1)
FixSpriteToScreen(6,1)
FixSpriteToScreen(7,1)
rem *** Zoom/Scroll using buttons ***
zoom# = 1
offsetX# = 0
offsetY# = 0
do
    if GetPointerState() = 1
        select GetSpriteHit(ScreenToWorldX(
            ↵GetPointerX()), ScreenToWorldY(
            ↵GetPointerY()))
            case 2:
                zoom# = zoom# + 0.1
            endcase
            case 3:
                zoom# = zoom# - 0.1
            endcase
            case 4:
                offsetX# = offsetX# - 1
            endcase
            case 5:
                offsetX# = offsetX# + 1
            endcase
            case 6:
                offsetY# = offsetY# - 1
            endcase
            case 7:
                offsetY# = offsetY# + 1
            endcase
        endselect
    endif

```

```

rem *** Set zoom factor ***
SetViewZoom(zoom#)
rem *** Set Offset factor ***
SetViewOffset(offsetX#,offsetY#)
Sync()
Sleep(100)
loop

```

Activity 19.12

Modified code for *ZoomScroll*:

```

rem *** Zooming and Scrolling ***

rem *** Load images ***
LoadImage(1,"AilsaCraig.jpg")
LoadImage(2,"In.png")
LoadImage(3,"Out.png")
LoadImage(4,"Left.png")
LoadImage(5,"Right.png")
LoadImage(6,"Up.png")
LoadImage(7,"Down.png")
rem *** Create main image ***
CreateSprite(1,1)
SetSpriteSize(1,100,-1)
rem *** Create text object ***
CreateText(1,"")
rem *** Create zoom in and zoom out buttons ***
CreateSprite(2,2)
SetSpriteSize(2,12,-1)
SetSpritePosition(2,85,92)
CreateSprite(3,3)
SetSpriteSize(3,12,-1)
SetSpritePosition(3,2,92)
rem *** Create scroll buttons ***
CreateSprite(4,4)
SetSpriteSize(4,10,-1)
SetSpritePosition(4,40,85)
CreateSprite(5,5)
SetSpriteSize(5,10,-1)
SetSpritePosition(5,50,85)
CreateSprite(6,6)
SetSpriteSize(6,10,-1)
SetSpritePosition(6,45,79)
CreateSprite(7,7)
SetSpriteSize(7,10,-1)
SetSpritePosition(7,45,91.5)
rem *** Fix buttons to screen ***
FixSpriteToScreen(2,1)
FixSpriteToScreen(3,1)
FixSpriteToScreen(4,1)
FixSpriteToScreen(5,1)
FixSpriteToScreen(6,1)
FixSpriteToScreen(7,1)
rem *** Fix text to screen ***
FixTextToScreen(1,1)
rem *** Zoom/Scroll using buttons ***
zoom# = 1
offsetX# = 0
offsetY# = 0
do
    if GetPointerState() = 1
        select GetSpriteHit(ScreenToWorldX(
            ↵GetPointerX()), ScreenToWorldY(
            ↵GetPointerY()))
            case 2:
                zoom# = zoom# + 0.1
            endcase
            case 3:
                zoom# = zoom# - 0.1
            endcase
            case 4:
                offsetX# = offsetX# - 1
            endcase
            case 5:
                offsetX# = offsetX# + 1
            endcase
            case 6:
                offsetY# = offsetY# - 1
            endcase
            case 7:
                offsetY# = offsetY# + 1
            endcase
        endselect
    endif
    rem *** Set zoom factor ***
    SetViewZoom(zoom#)
    rem *** Set Offset factor ***
    SetViewOffset(offsetX#,offsetY#)

```



```

rem *** Display offsets ***
SetTextString(1,"X:"+Str(GetViewOffsetX(),1)+
    " Y:"+Str(GetViewOffsetY(),1))
Sync()
Sleep(100)
loop

```

Activity 19.13

To display the mouse position in world coordinates requires the lines

```

rem *** Display offsets ***
SetTextString(1,"X:"+Str(GetViewOffsetX(),1)+
    " Y:"+Str(GetViewOffsetY(),1))

```

to be changed to

```

rem *** Display mouse position in world coords ***
SetTextString(1,"World X:"+Str(ScreenToWorldX(
    GetPointerX(),1)+" World Y:"+Str(ScreenToWorldY(
    GetPointerY(),1))

```

Activity 19.14

When you attempt a second scroll, the image jumps back to its original position.

Activity 19.15

To correct the scrolling, change the first `case` option to

```

case 1: // Main image sprite
rem *** IF mouse just pressed, save start
    position ***
    if pressed = 1
        x# = GetPointerX()
        y# = GetPointerY()
        xoffsetatstart# = GetViewOffsetX()
        yoffsetatstart# = GetViewOffsetY()
    else
rem *** else record latest position ***
        newx# = GetPointerX()
        newy# = GetPointerY()
    endif
endcase

```

and line

```

SetViewOffset(x#-newx#,y#-newy#)

```

to

```

SetViewOffset(xoffsetatstart#+x#-newx#,
    yoffsetatstart#+y#-newy#)

```

After zooming, the distance the image moves when scrolling no longer matches the distance moved by the pointer.

Activity 19.16

To make scrolling work correctly when the magnification is other than 1, change the line

```

SetViewOffset(xoffsetatstart#+x#-newx#,
    yoffsetatstart#+y#-newy#)

```

to

```

SetViewOffset(xoffsetatstart#+(x#-newx#)/zoom#,
    yoffsetatstart#+(y#-newy#)/zoom#)

```

Activity 19.17

To have the program count only recognised touches, change the parameter for `GetRawTouchCount()` to 0.

This option increases the time taken for a touch to register in the count.

Activity 19.18

To include unrecognised touches, change the parameter for `GetRawFirstTouchEvent()` to 1.

Activity 19.19

Modified code for *Touch02*:

```

rem *** Touch Times ***

rem *** Create text ***
CreateText(1,"")
do
    rem *** Get first touch ***
    id = GetRawFirstTouchEvent(0)
    while id <> 0
        rem *** Display its ID and time ***
        SetTextString(1,Str(id)+" "
            +Str(GetRawTouchTime(id)))
        rem *** Get next touch ***
        id = GetRawNextTouchEvent()
    endwhile
    Sync()
loop

```

Activity 19.20

No solution required.

Activity 19.21

No solution required.

Activity 19.22

When the `Sync()` statement is replaced with calls to `Update()`, `Render()` and `Swap()`, the movement of the ball seems less smooth.

Activity 19.23

No solution required.

Physics

In this Chapter:

- ☐ Using AGK's Physics Engine
- ☐ Giving a Sprite Velocity
- ☐ Setting a Sprite's Bounce Factor
- ☐ Adding Spin
- ☐ Adjusting Friction
- ☐ Setting Mass
- ☐ Detecting Collisions
- ☐ Bounding Boxes for Physics
- ☐ Physics and Ray Casting
- ☐ Joints
- ☐ Physics Debugging

Sprite Physics - 1

Introduction

So far we have had to write all the code required to make a sprite move or react to a collision. Sometimes that can require a considerable amount of effort and code. However, the good news is that AGK comes with a built-in physics engine.

What is a physics engine? In effect, it is a set of commands that, when applied to a sprite, will allow that sprite to react in an apparently natural way to gravity, friction, collisions and various other natural laws of physics.

In this chapter we cover all the physics commands available in AGK and demonstrate what effect they have on sprite behaviour.

Basic Physic Statements

SetSpritePhysicsOn()

All that is required to have a sprite react to the AGK's built-in physics is to apply the `SetSpritePhysicsOn()` statement to that sprite. This allows us to setup the sprite as one of three basic types:

- **A static object.** These objects never move, but dynamic objects will react to a collision with a static object. Typical real-world static objects would be items such as pavements, buildings, and mountains.
- **A dynamic object.** These objects move, react to gravity, friction, collisions, etc. Most small man-made objects in the real world would fall into this category.
- **A kinematic object.** A kinematic object is one which moves along a trajectory but will not deviate from that path and is not affected by other objects. The closest to a real-world object would be a meteorite or Superman!

FIG-20.1

SetSpritePhysicsOn()

The format for the `SetSpritePhysicsOn()` statement is shown in FIG-20.1.

`SetSpritePhysicsOn ((id , imode)`

where:

id is an integer value giving the ID of a previously created sprite.

imode is an integer value (1 to 3) specifying the type of physics object to be implemented (1: static; 2: dynamic; 3: kinematic).

FIG-20.2

A Dynamic Object

The program in FIG-20.2 demonstrates the use of physics to create a falling ball.

```
rem *** Basic Physics ***
rem *** Load image ***
LoadImage(1,"Ball.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,6,-1)
SetSpritePosition(1,50,5)
```



FIG-20.2

(continued)

A Dynamic Object

```
rem *** Apply dynamic physics ***
SetSpritePhysicsOn(1,2)
do
    Sync()
loop
```

► All projects in this chapter should be sized as 480 x 480 unless stated otherwise.

Activity 20.1

Start a new project called *Physics01* and implement the code given in FIG-20.2. Copy *AGKDownloads/Chapter20/Ball.png* to the *media* folder.

Set the window size to 480 x 480 and run the program.

What happens to the ball as it falls? What happens when the ball hits the bottom of the app window?

Modify the program so that the sprite is a static object. What happens when you run the program?

Modify the program to make the sprite a kinematic object. What happens when you run the program?

SetSpritePhysicsVelocity()

In the last Activity, there was no apparent difference between the static and kinematic modes. This is because no velocity had been specified for the kinematic sprite. To set a velocity, we need to use the `SetSpritePhysicsVelocity()` statement (see FIG-20.3).

FIG-20.3

SetSpritePhysicsVelocity()

`SetSpritePhysicsVelocity` (`id` , `vx` , `vy`)

where:

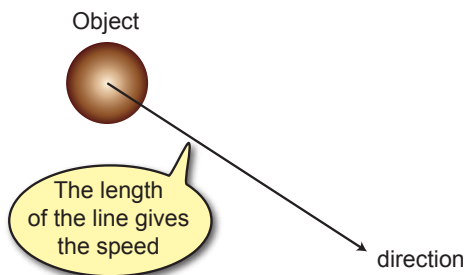
id is an integer value giving the ID of a previously created sprite.

vx,vy are real numbers giving the x and y components of the velocity to be applied to the sprite.

Velocity defines both speed and direction. In physics it is usually depicted graphically as an arrowed line (see FIG-20.4).

FIG-20.4

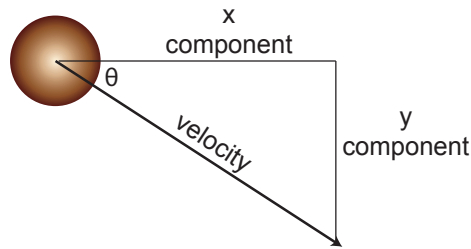
Velocity Visualised



To split the velocity into separate x, y components, we just need a simple piece of trigonometry (see FIG-20.5).

FIG-20.5

Velocity's X and Y
Components



If we know the direction of the velocity as an angle (say θ), the simplest way of calculating the x and y components is to use

$$x \text{ component} = \cos(\theta)$$

$$y \text{ component} = \sin(\theta)$$

and set the speed by multiplying these terms by the relevant amount. For example,

$$5 * \cos(\theta)$$

$$5 * \sin(\theta)$$

Activity 20.2

Modify *Physics01* so that the kinematic ball is moving at a speed of 15 at an angle of 60° . Test and save your project.

You can apply the `SetSpritePhysicsVelocity()` statement to a dynamic object, but this will interrupt the natural velocity assigned to the object by gravity, collisions, etc. Of course, one reason you might wish to do this is to simulate an engine burn on a spacecraft.

GetSpritePhysicsVelocityX() and GetSpritePhysicsVelocityY()

You can retrieve the x and y components of an object's velocity using the `GetSpritePhysicsVelocityX()` and `GetSpritePhysicsVelocityY()` statements. These statements have the format shown in FIG-20.6.

FIG-20.6

`GetSpritePhysicsVelocityX()`
`GetSpritePhysicsVelocityY()`

```
GetSpritePhysicsVelocityX ( id )
GetSpritePhysicsVelocityY ( id )
```

where:

id is an integer value giving the ID of a previously created sprite.

The program in FIG-20.7 is a variation on the earlier falling ball example, but this time the y component of the ball's velocity is displayed as it falls. No x component is displayed since the ball is falling vertically.

FIG-20.7

Displaying Velocity

```
rem *** Basic Physics ***
rem *** Load image ***
LoadImage(1,"Ball.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,6,-1)
```



FIG-20.7

(continued)

Displaying Velocity

```

SetSpritePosition(1,50,5)

rem *** Apply dynamic physics ***
SetSpritePhysicsOn(1,2)

rem *** Create text object ***
CreateText(1,"")

do
    rem *** Update text ***
    SetTextString(1,"Ball velocity Y : "
    +Str(GetSpritePhysicsVelocityY(1))
    Sync()
loop

```

Activity 20.3

Modify *Physics01* to match the code given above. Test and save your project.

SetSpritePhysicsOff()

You can return a sprite to a normal, non-physics-influenced sprite by switching off physics for that sprite. This requires the use of the `SetSpritePhysicsOff()` statement (see FIG-20.8).

FIG-20.8

SetSpritePhysicsOff()

where:

id is an integer value giving the ID of a previously created sprite.

If you want to change the nature of a sprite from, say, dynamic to static, you need to start by switching off the sprite's physics and then reactivating it in a different mode using `SetSpritePhysicsOn()`.

SetSpritePhysicsDelete()

If you switch off a sprite's physics, turning it back on will restore all the previous settings (assuming you create the sprite as the same type of physics object). However, if you want to go further than this, you can turn off a sprite's physics and remove all the previous settings using the `SetSpritePhysicsDelete()` statement (see FIG-20.9).

FIG-20.9

SetSpritePhysicsDelete()

where:

id is an integer value giving the ID of a previously created sprite.

SetSpritePhysicsRestitution()

When the ball dropped in our previous programming examples, we saw it bounce slightly as it hit the "floor" of the app. Obviously, some objects bounce to a greater or lesser degree in the real world. For example, we would expect a rubber ball to

bounce more than one made of iron.

The “bounciness” of an object can be set using the `SetSpritePhysicsRestitution()` statement (see FIG-20.10).

FIG-20.10

`SetSpritePhysicsRestitution()`

`SetSpritePhysicsRestitution (id , fbounce)`

where:

id is an integer value giving the ID of a previously created sprite.

fbounce is a real number (0 to 1) which defines the bounciness of the object. 0: no bounce; 1 : full bounce.

Activity 20.4

Modify *Physics01* to give the ball a bounce factor of 0.75. Test and save your project.

SetSpritePhysicsAngularVelocity()

Adding a spinning effect to an object can be achieved using the `SetSpritePhysicsAngularVelocity()` statement. This has the format shown in FIG-20.11.

FIG-20.11

`SetSpritePhysicsAngular
Velocity()`

`SetSpritePhysicsAngularVelocity (id , fav)`

where:

id is an integer value giving the ID of a previously created sprite.

fav is a real number giving the angular velocity to be applied. This is the angle by which the object rotates in each time frame. The larger the number, the faster the spin.

Angular velocity is best applied to a dynamic object when it is first created or to a kinematic object. The program in FIG-20.12 demonstrates a spinning kinematic rectangle.

FIG-20.12

Using Angular Velocity

```
rem *** Angular Velocity ***

rem *** Load image ***
LoadImage(1,"Tile.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,10,10)

rem *** Apply kinematic physics ***
SetSpritePhysicsOn(1,3)
rem *** Apply angular velocity ***
SetSpritePhysicsAngularVelocity(1,1)

do
    Sync()
loop
```


Activity 20.5

Start a new project called *Physics02* and implement the code given in FIG-20.12. Copy *Tile.png* from *AGKDownloads/Chapter20* to the *media* folder.

After you have run the program, change the sprite to a dynamic object and the angular velocity to 50. How does this affect the object?

Test and save your project.

SetSpritePhysicsAngularDamping()

In the real world, spinning objects normally lose their spin because of frictional forces. To create the same effect in our virtual world, we can use the `SetSpritePhysicsAngularDamping()` statement (see FIG-20.13).

FIG-20.13

SetSpritePhysicsAngular
Damping()

`SetSpritePhysicsAngularDamping ((id , fdamp)`

where:

id is an integer value giving the ID of a previously created sprite.

fdamp is a real number (0 to 2) giving the angular damping to be applied. 0: no dampening; 2: rotation stops almost immediately.

Damping cannot be applied to a kinematic object.

Activity 20.6

Add a damping factor of 0.7 to the sprite in *Physics02* and observe how this affects the tile.

SetSpritePhysicsTorque()

Torque is the term used for a turning force applied to an object. When you turn a nut with a wrench, you are applying torque to the nut. The `SetSpritePhysicsTorque()` command applies a turning force to an object. Although this will cause an object to spin, and hence you might be tempted to think that it performs the same purpose as `SetSpritePhysicsAngularVelocity()`, the `SetSpritePhysicsTorque()` statement creates a more realistic effect on dynamic objects, since it takes into account the current state of the object. Torque forces are cumulative, so applying the same force continually will speed up the rotation of the object to which it is being applied.

FIG-20.14

SetSpritePhysicsTorque()

The `SetSpritePhysicsTorque()` statement has the format shown in FIG-20.14.

`SetSpritePhysicsTorque ((id , ftorque)`

where:

id is an integer value giving the ID of a previously created sprite.

ftorque is a real number giving the angular force to be applied. This force is cumulative. A negative figure will cause the object to spin in the opposite (counter-clockwise) direction.

The program in FIG-20.15 makes use of torque rather than setting angular momentum to spin an object.

FIG-20.15

Using Torque

```
rem *** Torque ***

rem *** Load image ***
LoadImage(1,"Tile.png")

rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,40,10)

rem *** Apply dynamic physics ***
SetSpritePhysicsOn(1,2)
do
    rem *** Apply torque ***
    SetSpritePhysicsTorque(1,20)
    Sync()
loop
```

Activity 20.7

Modify *Physics02* to match the code in FIG-20.15.

Run the program and observe the effect.

Change the torque from 20 to 200. How does this affect the behaviour of the sprite?

SetSpritePhysicsAngularImpulse()

The `SetSpritePhysicsAngularImpulse()` statement has a similar effect to applying torque but it is designed to be executed only once, not continuously. Use this command if you want to simulate an object that has been given added angular velocity because of a collision.

FIG-20.16

SetSpritePhysicsAngular
↳ Impulse()

`SetSpritePhysicsAngularImpulse` ((`id` , `fimpulse`))

where:

- | | |
|-----------------|---|
| id | is an integer value giving the ID of a previously created sprite. |
| fimpulse | is a real number giving the angular force to be applied. This force is cumulative. A negative figure will cause the object to spin in the opposite (counter-clockwise) direction. |

The effect created by the `SetSpritePhysicsAngularImpulse()` statement is equivalent to a torque of the same force applied for one second.

Both torque and impulse forces have a greater effect on the angular velocity of a smaller object than on a larger one for the same amount of force.

GetSpritePhysicsAngularVelocity()

The rate at which an object is spinning can be discovered using the `GetSpritePhysicsAngularVelocity()` statement which has the format shown in FIG-20.17.

FIG-20.17

GetSpritePhysicsAngular
Velocity()

float `GetSpritePhysicsAngularVelocity` ((`id`))

where:

id is an integer value giving the ID of a previously created sprite.

Activity 20.8

Modify *Physics02* so that the angular velocity of the sprite is continuously displayed.

Save your project.

SetSpritePhysicsCanRotate()

You can switch off a dynamic object's ability to rotate using the `SetSpritePhysicsCanRotate()` statement (see FIG-20.18).

FIG-20.18

SetSpritePhysicsCanRotate()

`SetSpritePhysicsCanRotate` ((`id`) , `iflag`)

where:

id is an integer value giving the ID of a previously created sprite.

iflag is an integer value (0 or 1) which determines if a dynamic sprite is allowed to rotate. 0: no rotation allowed; 1: rotation allowed.

The main use of this statement is to stop an object being given angular momentum when struck by some other object.

Activity 20.9

Modify *Physics02* by switching off the sprite's ability to rotate immediately after the `SetSpritePhysicsOn()` statement.

How is the sprite affected?

Save your project.

SetSpritePhysicsForce()

As well as being able to apply forces that set an object spinning, we can also apply a force to move an object to the side, up, or down (although gravity will do the down option for us).

When we do this, the direction and strength of that force will affect how the object reacts to the force (see FIG-20.19).

FIG-20.19

Force Explained

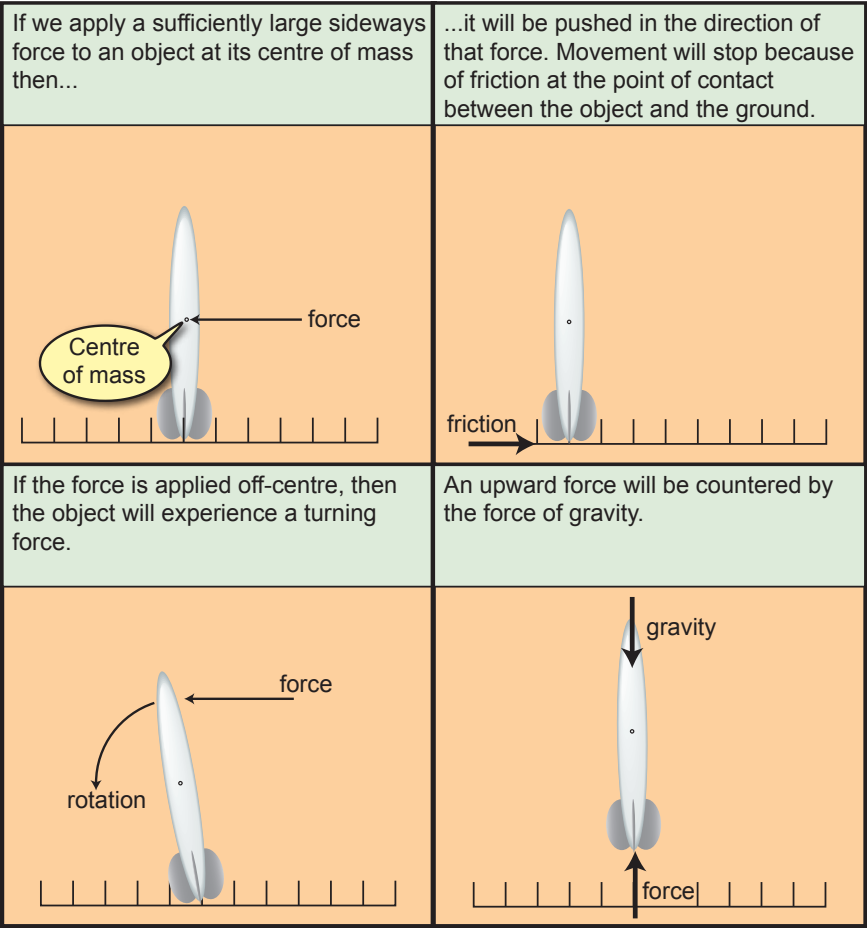


FIG-20.20

SetSpritePhysicsForce()

SetSpritePhysicsForce ((id , x , y , vx , vy))

where:

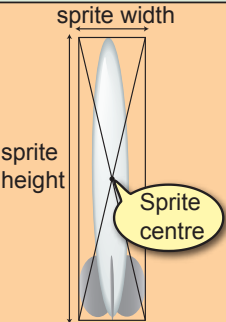
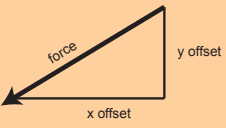
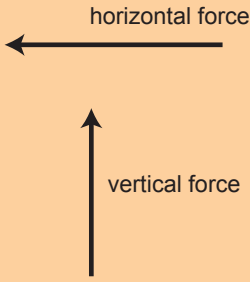
- id** is an integer value giving the ID of a previously created sprite.
- x,y** are a pair of real numbers giving the coordinates of any point along the line of force. This value is used to identify the exact position acted on by the force.
- vx** is a real number giving the *x* offset of the force. A positive value implies a force pushing to the right; a negative one creates a push to the left.
- vy** is a real number giving the *y* offset of the force. A positive value pushes down; a negative value pushes up.

These last two values define the strength of the force.

FIG-20.21 explains the values required when defining a force.

FIG-20.21

Calculating Force Vectors

The centre of mass (COM) of a sprite is assumed to be at the centre of the sprite.	The coordinates of the centre of mass can be calculated as:
	$X_{\text{com}} = \text{GetSpriteX}(\text{spr}) + \text{GetSpriteWidth}(\text{spr}) / 2.0$ $y_{\text{com}} = \text{GetSpriteY}(\text{spr}) + \text{GetSpriteHeight}(\text{spr}) / 2.0$
	These points can be used as the coordinates on the line of force. This ensures there is no turning force applied to the object.
The force offsets are calculated by a simple Pythagorean formula. Larger values mean a stronger force.	A horizontal force has no y offset; a vertical force has no x offset.
	

The force applied by `SetSpritePhysicsForce()` is applied only once, so if you want the force to be applied continually, the statement must be placed within a loop.

The program in FIG-20.22 creates a pushing force under the spaceship sprite, causing it to lift into the air.

FIG-20.22

Using a Force Vector

```

rem *** Force Upwards ***

rem *** Load image ***
LoadImage(1,"Rocketship.png")

rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,-1,20)
SetSpritePosition(1,45,80)

rem *** Turn on physics for sprite ***
SetSpritePhysicsOn(1,2)

do
    rem *** Apply upward force through centre of craft ***
    SetSpritePhysicsForce(1,GetSpriteX(1)+GetSpriteWidth(1)/2.0,
        GetSpriteY(1)+GetSpriteHeight(1)/2.0,0,-200)
    Sync()
loop

```

Activity 20.10

Create a new project called *Physics03* and implement the code in FIG-20.22. Copy *AGKDownload/Chapter20/Rocketship.png* to the *media* folder.

Test and save your project. What happens when the craft reaches the top of the app window?

Modify the program to change the vertical force to a horizontal force to the right, passing through the centre of the craft.

What effect does this have on the craft?

Move the `SetSpritePhysicsForce()` statement so that it is positioned before the `do` statement. Return the force to an upward one and change the `vy` value to -20000. How does this affect the craft?

SetSpritePhysicsImpulse()

The `SetSpritePhysicsImpulse()` statement is similar to `SetSpritePhysicsForce()` but this time the force is applied for the equivalent of one second. This command is primarily intended to be used when a one-off force is required such as an object being struck momentarily by a laser beam.

FIG-20.23

The format for `SetSpritePhysicsImpulse()` is given in FIG-20.23.

`SetSpritePhysicsImpulse()` `SetSpritePhysicsImpulse` `(` `id` `,` `x` `,` `y` `,` `vx` `,` `vy` `)`

where:

- id** is an integer value giving the ID of a previously created sprite.
- x,y** are a pair of real numbers giving the coordinates of any point along the line of force. This value is used to identify the exact position of the force.
- vx** is a real number giving the *x* offset of the force. A positive value implies a force pushing to the right; a negative one creates a push to the left.
- vy** is a real number giving the *y* offset of the force. A positive value pushes down; a negative value pushes up.

Activity 20.11

Modify *Physics03*, reducing the `vy` value in the `SetSpritePhysicsForce()` statement to -2000).

How far does the spacecraft rise with this setting?

Replace the `SetSpritePhysicsForce()` statement with a `SetSpritePhysicsImpulse()` statement using the same parameters.

How far does the spacecraft rise this time?

SetSpritePhysicsMass()

When a force is applied to an object, how that object reacts depends on the strength of the force and the mass of the object. AGK assigns a default mass to a sprite based on the area of the sprite. However, this is not always appropriate. For example, an object constructed from Styrofoam may be the same size as an object made from steel, but the mass of the two objects will be widely different.

You can define the mass of an object using the `SetSpritePhysicsMass()` statement whose syntax is given in FIG-20.24.

FIG-20.24

SetSpritePhysicsMass()

`SetSpritePhysicsMass` ((`id` , `fkilos`)

where:

id is an integer value giving the ID of a previously created sprite.

fkilos is a real number giving the new mass in kilograms.

GetSpritePhysicsMass()

To determine the mass currently assigned to an object, call `GetSpritePhysicsMass()` (see FIG-20.25) which returns a specified object's mass in kilograms.

FIG-20.25

GetSpritePhysicsMass()

float `GetSpritePhysicsMass` ((`id`)

where:

id is an integer value giving the ID of a previously created sprite.

Activity 20.12

Modify *Physics03*, so that the mass of the spacecraft is displayed.

Modify the program again so that the spacecraft is ten times its original weight.

How does this affect the craft's movement?

SetSpritePhysicsFriction()

When two surfaces move over each other, friction is created which impedes movement, eventually bringing the moving object to a halt.

The degree of friction created by a sprite can be set using the `SetSpritePhysicsFriction()` statement (see FIG-20.26).

FIG-20.26

SetSpritePhysicsFriction()

`SetSpritePhysicsFriction` ((`id` , `friction`)

where:

id is an integer value giving the ID of a previously created sprite.

friction is a real number (0 to 1) specifying the friction created by the sprite. 0: no friction; 1 : full friction. The default friction value is around 0.3.

Activity 20.13

Modify *Physics03*, so that the force is again horizontal and to the left then run the program. What happens to the craft?

Set friction of the spacecraft to 0.25. How does this affect the craft's movement?

What is the lowest value of friction (to the nearest one hundredth) that causes the craft to topple?

Set friction to zero. What affect does this have on the craft?

Save your project.

SetSpritePhysicsDamping()

With no friction, the craft slides over the surface at a constant speed until it hits the edge of the window. In real life, no matter how low the surface friction, the craft would still stop, slowed by air resistance. To simulate air resistance (or water resistance if our sprite was a boat) we can make use of `SetSpritePhysicsDamping()` (see FIG-20.27).

FIG-20.27

SetSpritePhysicsDamping()

`SetSpritePhysicsDamping` (`id` , `fdamp`)

where:

id is an integer value giving the ID of a previously created sprite.

fdamp is a real number (0 to 1) specifying the damping factor to be applied. 0: no damping (the default); 1 : full damping (movement stops almost immediately).

Activity 20.14

Modify *Physics03*, so that a damping force of 0.2 is applied?

How does this affect the movement of the craft? Save your project.

SetSpritePhysicsCOM()

By default, a sprite's centre of mass (COM) is at the centre of the image (as was explained earlier) but you can move this to another position which is more appropriate for the object your sprite represents. For example, a railway carriage would have a lower centre of gravity because the wheels and chassis are much heavier than the walls and roof of the compartment area.

FIG-20.28

To shift a sprite's centre of gravity, use `SetSpritePhysicsCOM()` (see FIG-20.28).

SetSpritePhysicsCOM()

`SetSpritePhysicsCOM` (`id` , `x` , `y`)

where:

id is an integer value giving the ID of a previously created sprite.

x,y are real values giving the position of the new centre of mass relative to the sprite's offset values.

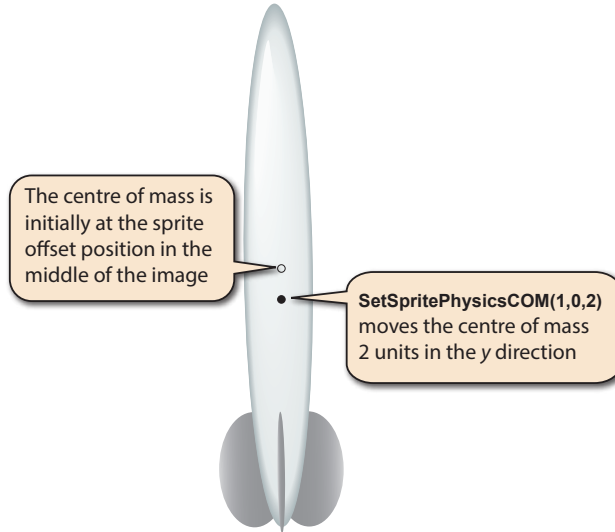
For example, if we were to use the statement

```
SetSpritePhysicsCOM(1,0,2)
```

FIG-20.29

the centre of mass would be moved as shown in FIG-20.29.

Changing the Centre of Mass



CalculateSpritePhysicsCOM()

FIG-20.30

You can also leave AGK to recalculate the centre of mass using the `CalculateSpritePhysicsCOM()` statement (see FIG-20.30).

CalculateSpritePhysics
COM()

```
CalculateSpritePhysicsCOM ( id )
```

where:

id is an integer value giving the ID of a previously created sprite.

If you have overridden the centre of mass position using `SetSpritePhysicsCOM()`, `CalculateSpritePhysicsCOM()` will reset its position. You may also wish to call `CalculateSpritePhysicsCOM()` if you have assigned a new bounding area to your sprite (see later for physics-activated sprite bounding area assignments).

SetSpritePhysicsIsSensor()

If you want a physics sprite to detect collisions but not react to them in anyway and also not cause the colliding sprite to be affected, then you can set the sprite to be a sensor using the `SetSpritePhysicsIsSensor()` statement (see FIG-20.31).

FIG-20.31

SetSpritePhysicsIsSensor()

```
SetSpritePhysicsIsSensor ( id , iflag )
```

where:

id is an integer value giving the ID of a previously created sprite.

iflag is an integer value (0 or 1) which is used to set the sprite to sensor mode (0) or return it to normal mode (1).

Physics Collisions

So far, we have made use of a single physics-based sprite in all of the examples. But when more than one sprite exists, then the moving sprites are likely to collide. Unlike standard sprites, physics-activated sprites will deal with collisions automatically. The program in FIG-20.32 demonstrates this by showing a ball colliding with a spinning bat.

FIG-20.32

Collisions in Physics

```
rem *** Physics collisions ***

rem *** Load images ***
LoadImage(1,"Bat.png")
LoadImage(2,"Ball.png")

rem *** Set up bat ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
SetSpritePosition(1,44,48)

rem *** Make bat kinematic ***
SetSpritePhysicsOn(1,3)

rem *** Spin bat ***
SetSpritePhysicsAngularVelocity(1,100)

rem *** Set up ball ***
CreateSprite(2,2)
SetSpriteSize(2,6,-1)
SetSpritePosition(2,47,0)

rem *** Make ball dynamic ***
SetSpritePhysicsOn(2,2)
do
    Sync()
loop
```

Activity 20.15

Start a new project called *Physics04* and implement the code given above. Copy *Ball.png* and *Bat.png* from *AGKDownloads/Chapter20* into the *media* folder. Test and save your project.

GetPhysicsCollision()

Even though AGK automatically handles collisions, a program will sometimes need to be aware of a collision taking place. This can be achieved using the `GetPhysicsCollision()` statement (see FIG-20.33).

FIG-20.33

`GetPhysicsCollision()` integer `GetPhysicsCollision` (`id1` , `id2`)

where:

id1,id2 are integer values giving the IDs of previously created sprites.

If the two sprites have collided, then the function returns 1, otherwise zero is returned.

Activity 20.16

Modify *Physics04* to display the word “Hit” when the sprites collide. Test and save your project.

GetPhysicsCollisionX() and GetPhysicsCollisionY()

After detecting a collision using `GetSpritePhysicsCollision()`, you can determine the exact position of the collision (relative to the first sprite identified in `GetSpritePhysicsCollision()`) using the `GetPhysicsCollisionX()` and `GetPhysicsCollisionY()` statements (see FIG-20.34).

FIG-20.34

```
GetPhysicsCollisionX()    float  GetPhysicsCollisionX ( )
GetPhysicsCollisionY()    float  GetPhysicsCollisionY ( )
```

Activity 20.17

Modify *Physics04* so that the coordinates of the collision are displayed in place of the word “Hit”. Test and save your project.

GetPhysicsCollisionWorldX() and GetPhysicsCollisionWorldY()

If it would be more useful to have the collision’s coordinates specified in world coordinates, then you can use the `GetPhysicsCollisionWorldX()` and `GetPhysicsCollisionWorldY()` statements (see FIG-20.35).

FIG-20.35

```
GetPhysicsCollisionWorldX()    float  GetPhysicsCollisionWorldX ( )
GetPhysicsCollisionWorldY()    float  GetPhysicsCollisionWorldY ( )
```

Activity 20.18

Modify *Physics04* so that the world coordinates are used to display the position of the collision. Test and save your project.

SetSpritePhysicsIsBullet()

If a sprite is moving very quickly (typically this would be a bullet or a missile) then you can force AGK to do more frequent checks for collisions using the `SetSpritePhysicsIsBullet()` statement. Without this, collision detection for fast moving items may not be accurate. The format of the `SetSpritePhysicsIsBullet()` statement is shown in FIG-20.36.

FIG-20.36

```
SetSpritePhysicsIsBullet(    SetSpritePhysicsIsBullet ( id , iflag )
```

where:

id is an integer value giving the ID of a previously created sprite.

iflag is an integer value (0: normal collision checking 1: increased collision checking).

Physics Sprite Shapes

In the last chapter we saw how changing the bounding shape for a sprite could lead to more accurate collision detection. The same type of option is also available for sprites for which the physics option has been switched on. However, creating a bounding box for physics objects requires new statements.

By default, every physics object is assigned a rectangular bounding box. This is used not only to detect collisions, but also to handle other calculations such as friction and bounce.

SetPhysicsDebugOn()

We can make the bounding box visible by calling `SetPhysicsDebugOn()` (see FIG-20.37).

FIG-20.37

SetPhysicsDebugOn()

SetPhysicsDebugOn (())

Activity 20.19

Modify *Physics04* so that physics debug is on during the running of the program.

SetPhysicsDebugOff()

Should you want the debug option switched off at a later stage in a program's execution, you can use the `SetPhysicsDebugOff()` statement (see FIG-20.38).

FIG-20.38

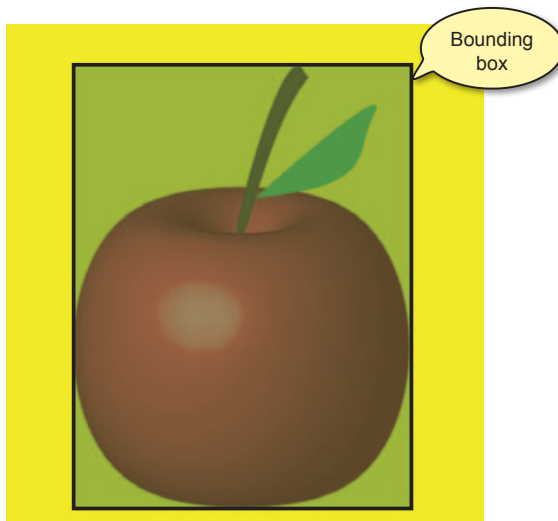
SetPhysicsDebugOff()

SetPhysicsDebugOff (())

FIG-20.39

A Default Bounding Box

FIG-20.39 shows the default bounding box assigned to an apple-shaped sprite.

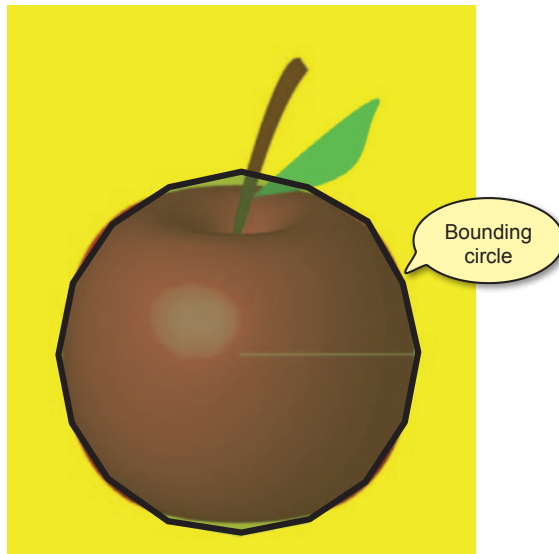


As you can see, this is far from ideal. We could replace the default bounding box by a circular bounding area around the main apple using `SetSpriteShapeCircle()` which would give the result shown in FIG-20.40.

FIG-20.40

Creating a Circular
Bounding Box

But the problem now is that the stalk and leaf remain outside the bounding area.



However, although standard sprites are limited to a single bounding area shape, physics sprites may be given additional shapes. If you do this, then the new bounding area will include all the shapes assigned to the sprite. You can only add these new shapes after physics has been switched on for the sprite.

AddSpriteShapeBox()

To add a rectangular bounding area to your physics sprite, use `AddSpriteShapeBox()` (see FIG-20.41).

FIG-20.41 AddSpriteShapeBox()

```
AddSpriteShapeBox ( ( id , x1 , y1 , x2 , y2 , fangle ) )
```

where

- id** is an integer value giving the ID of the sprite.
- x1,y1** are a pair of real values giving the coordinates of the top-left corner of the box.
- x2,y2** are a pair of real values giving the coordinates of the bottom-right corner of the box.
- fangle** is a real number giving the angle of rotation of the box in radians.

The box coordinates are measured relative to the top-left corner of the sprite to which the bounding box is being added and represent the position of the box before it is rotated.

AddSpriteShapePolygon()

To add a polygon to the bounding area of a physics sprite, use the `AddSpriteShapePolygon()` statement (see FIG-20.42).

FIG-20.42 AddSpriteShapePolygon()

AddSpriteShapePolygon ((id , inum , indx , x , y)

where

- id** is an integer value giving the ID of the sprite.
- inum** is an integer value (3 to 12) giving the number of vertices in the polygon.
- indx** is an integer value giving the index of this specific vertex. The first vertex has an index setting of zero.
- x,y** are a pair of real values giving the coordinates of the vertex. The coordinates are measured from the sprite's offset.

FIG-20.43 shows the apple after both a box and polygon have been added to the original bounding area.

FIG-20.43

A Sprite with Three
Bounding Areas

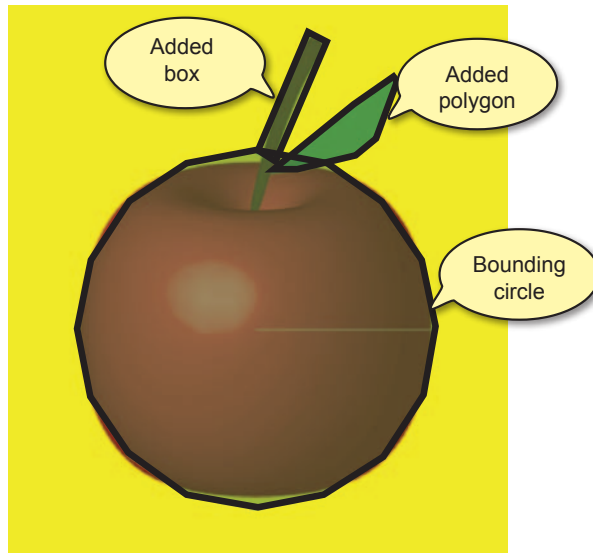


FIG-20.44

Creating Additional
Bounding Areas on a
Physics Sprite

The code required to produce the bounding area shown above is given in FIG-20.44.

```
rem *** Sprite Bounding Area ***

rem *** Vertex coordinates on polygon ***
dim verticesX#[11]=[20.43,14.37,2.22,6.68,14.07,17.10,18.01,20.94]
dim verticesY#[11]=[-22.49,-20.62,-11.27,-11.44,-13.04,-15.07,
↳-16.67,-21.98]

rem *** Number of vertices ***
global vcount = 8

rem *** Set background colour ***
SetClearColor(250,250,20)
Sync()

rem *** Load image ***
LoadImage(1,"Apple2.png")
```



FIG-20.44

(continued)

Creating Additional
Bounding Areas on a
Physics Sprite

► Additional
bounding areas will
affect the centre of
mass of a sprite.

```
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,50,-1)
SetSpritePositionByOffset(1,50,50)

rem *** Bound fruit ***
SetSpriteShapeCircle(1,0,7.5,25.5)

rem *** Make static physics object ***
SetSpritePhysicsOn(1,1)

rem *** Add box to apple stalk ***
AddSpriteShapeBox(1,28.5,-0.5,31.5,15,0.41)

rem *** Add polygon to apple leaf ***
for c = 0 to vcount-1
    AddSpriteShapePolygon(1,vcount,c,verticesX#[c],verticesY#[c])
next c

rem *** Switch on physics debug ***
SetPhysicsDebugOn()

do
    Sync()
loop
```

The values given in the code assume the app window is set to 1000x1200; for other windows sizes the values given would need to be changed.

Activity 20.20

Start a new project called *Physics05* and implement the code given in FIG-20.44. Copy *Apple2.png* to the *media* folder. Set the app window size to 1000x1200.

Test and save your project.

AddSpriteShapeCircle()

A circle can also be added to the bounding area of a physics sprite using **AddSpriteShapeCircle()** (see FIG-20.45).

FIG-20.45

AddSpriteShapeCircle()

AddSpriteShapeCircle (**id** , **x** , **y** , **fradius**)

where:

► Bounding areas added
to a sprite will not resize
with the sprite.

id is an integer value giving the ID previously assigned to the sprite.

x,y are a pair of real values giving the coordinates of the centre of the bounding circle. These are measured relative to the sprite's offset.

fradius is a real number giving the radius of the bounding circle.

ClearSpriteShapes()

The shapes that have been added to a sprite can be removed using the `ClearSpriteShapes()` statement (see FIG-20.46).

FIG-20.46

ClearSpriteShapes()

`ClearSpriteShapes (id)`

where

id is an integer value giving the ID of the sprite.

SetPhysicsMaxPolygonPoints()

You can limit the maximum number of vertices on any future bounding polygon using `SetPhysicsMaxPolygonPoints()` (see FIG-20.47).

FIG-20.47

SetPhysicsMaxPolyPoints()

`SetPhysicsMaxPolyPoints (ipnts)`

where

ipnts is an integer (2 to 12) specifying the maximum vertices on any future bounding polygons for physics-activated sprites.

Summary

- The physics engine embedded within AGK automatically simulates various aspects of the physical world such as velocity, friction, gravity and collisions.
- To activate physics for a specific sprite use `SetSpritePhysicsOn()`.
- A physics-activated sprite can be assigned to one of three categories:
 - static - unaffected by physics but other sprites react to colliding with a static sprite.
 - dynamic - reacts to all physics.
 - kinematic - can be assigned a velocity but is otherwise unaffected by the physics.
- Use `SetSpritePhysicsOff()` to deactivate a sprite's physics.
- After switching off a sprite's physics, it can be reactivated with its previous settings.
- Use `SetSpritePhysicsDelete()` to deactivate a sprite's physics and remove all current physics settings.
- Use `SetSpritePhysicsVelocity()` to assign a velocity to a sprite.
- Use `GetSpritePhysicsVelocityX()` and `GetSpritePhysicsVelocityY()` to determine the *x* and *y* components of a velocity.
- Use `SetSpritePhysicsRestitution()` to set the bounciness of a sprite.
- Use `SetSpritePhysicsAngularVelocity()` to add a spin to a sprite.
- Use `SetSpritePhysicsAngularDamping()` to adjust the damping effect on a sprite's spin.
- Use `SetSpritePhysicsTorque()` to add an accumulative spin effect to a sprite.

- Use `SetSpritePhysicsAngularImpulse()` to apply a one-off spin force to a sprite.
- Use `GetSpritePhysicsAngularVelocity()` to determine how fast a sprite is spinning.
- Use `SetSpritePhysicsCanRotate()` to adjust a sprite's ability to rotate.
- Use `SetSpritePhysicsForce()` to apply a force to a sprite.
- Use `SetSpritePhysicsImpulse()` to apply a force for the equivalent of one second.
- All sprites are assigned a default mass based on their size.
- Use `SetSpritePhysicsMass()` to adjust the mass of a sprite.
- Use `GetSpritePhysicsMass()` to discover the mass of a sprite.
- Use `SetSpritePhysicsFriction()` to set the friction of a sprite.
- Use `SetSpritePhysicsDamping()` to adjust the damping factor on a sprite's movement.
- Use `SetSpritePhysicsCOM()` to adjust a sprite's centre of mass.
- Use `CalculateSpritePhysicsCOM()` to have AGK calculate a sprite's centre of mass.
- Use `SetSpritePhysicsIsSensor()` to have a sprite detect collisions but not react to them.
- Use `GetPhysicsCollision()` to detect if two specified sprites have collided.
- After detecting a collision using `GetPhysicsCollision()`, use `GetPhysicsCollisionX()` and `GetPhysicsCollisionY()` to find the coordinates of a collision relative to the position of the first sprite.
- Use `GetPhysicsCollisionWorldX()` and `GetPhysicsCollisionWorldY()` to discover the world coordinates of a collision.
- Use `SetSpritePhysicsIsBullet()` if a sprite is moving extremely fast in order to ensure all collisions are detected correctly.
- Use `SetPhysicsDebugOn()` to display the bounding area of a physics-enabled sprite.
- Use `SetPhysicsDebugOff()` to hide the bounding area of a sprite.
- Use `AddSpriteShapeBox()` to add a new bounding box to the already-existing bounding area of a physics-enabled sprite.
- Use `AddSpriteShapePolygon()` to add a new bounding polygon to the already-existing bounding area of a physics-enabled sprite.
- Use `AddSpriteShapeCircle()` to add a new bounding circle to the already-existing bounding area of a physics-enabled sprite.
- Use `ClearSpriteShapes()` to remove all extra bounding areas assigned to a sprite.
- Use `SetPhysicsMaxPolyPoints()` to set the maximum number of vertices allowed on all new bounding polygons.

Introduction

So far, almost all of the physics commands we have covered have adjusted the properties of individual sprites, but an additional set of physics commands are available which adjust the complete environment, impacting on all the physics-enabled sprites within a project. These physics commands are listed in this section.

General Statements

SetPhysicsScale()

The physics environment used by AGK assumes one linear unit of screen space is equal to one metre in the real world. If you are using the default percentage system for the app screen, then 1% of screen width or height represents 1 metre in the real world. This would mean that the screen space is assumed to be 100 metres by 100 metres (even if the app window is not square!). In fact, AGK automatically changes this scale so that the screen represents a 20 metre by 20 metre area.

We can use the `SetPhysicsScale()` statement to adjust this metres-to-screen relationship, making the screen space represent a smaller (or larger) area in the real world. The `SetPhysicsScale()` statement has the format shown in FIG-20.48.

FIG-20.48

SetPhysicsScale()

`SetPhysicsScale` (`fscale`)

where:

fscale is a real number giving the scale factor to be used when calculating the relationship between real-world metres and screen units. A value of 1 would mean the screen represents a 100 metre by 100 metre area. The default setting is 0.2 (20 metres by 20 metres).

This statement should be called before the physics system is switched on.

SetPhysicsGravity()

Using `SetPhysicsGravity()` allows you to set both the force and direction of gravitational acceleration. The format of the statement is given in FIG-20.49.

FIG-20.49

SetPhysicsGravity()

`SetPhysicsGravity` (`xoffset` , `yoffset`)

where:

xoffset is a real number giving the x offset for the gravity vector.

yoffset is a real number giving the y offset for the gravity vector.

Normally, gravity will be “down” towards the bottom of the screen, so the *xoffset* value would be zero in that case.

These values represent how far in screen units an object will fall in the first second, so to relate it to real world gravity (which is 9.8 metres/sec² on Earth) you need to

take into account the physic scale setting.

Using the lines

```
SetPhysicsScale(1)
SetPhysicsGravity(0,9.8)
```

would create normal Earth gravity.

But if the default scale value of 0.2 is being used, then to achieve Earth gravity, we would need to use the line

```
SetPhysicsGravity(0,49)
```

In general we can say that Earth gravity is achieved by the formula:

$$yoffset = 9.8 / \text{physics scale setting}$$

However, Earth gravity is the default setting irrespective of the physics scale used, so you only need to make use of the `SetPhysicsGravity()` statement if you require a different gravitational setting. For example, the moon's gravity causes an acceleration of 1.63 m/s² while that of Mars is 3.7 m/s².

The program in FIG-20.50 creates a standard falling body under Earth's gravity with the screen representing a 50 metre by 50 metre area.

FIG-20.50

Setting Scale and Gravity

```
rem *** Gravity ***

rem *** Load image ***
LoadImage(1,"Tile.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,6,-1)
SetSpritePosition(1,45,0)
rem *** Set physics scale ***
SetPhysicsScale(0.5)
rem *** Switch on physics ***
SetSpritePhysicsOn(1,2)
do
    Sync()
loop
```

Activity 20.21

Start a new project called *Gravity* and implement the code given in FIG-20.50.

You need to copy the file *Tile.png* from *AGKDownloads/Chapter20* into the *media* folder. Set the app window size to 768 by 768. Test your project.

Modify the code so that the app window represents a 100m x 100m area.

Modify the code to simulate lunar gravity.

Change the physics scale so that the app window represents a 25m x 25m area, making sure lunar gravity is retained. Save your project.

Of, course, if your program requires it, gravity can pull in other directions. A negative *y* offset value will cause objects to move upwards and adding an *x* offset value will cause sideways movement. For example, the line

```
SetPhysicsGravity(4,-4)
```

would cause objects to move towards the top right corner of the app window.

SetPhysicsWallBottom(), SetPhysicsWallLeft(), SetPhysicsWallRight() and SetPhysicsWallTop()

As you will have noticed in various examples, objects never move off-screen when using the physics engine. This is because perimeter “walls” are in place confining all movement to the area of the app window. We can control these four “walls”, switching them on and off using the statements `SetPhysicsWallBottom()`, `SetPhysicsWallLeft()`, `SetPhysicsWallRight()` and `SetPhysicsWallTop()`. The format for each of these statements is shown in FIG-20.51.

FIG-20.51

SetPhysicsWallBottom()

SetPhysicsWallBottom (iflag)

SetPhysicsWallTop()

SetPhysicsWallTop (iflag)

SetPhysicsWallLeft()

SetPhysicsWallLeft (iflag)

SetPhysicsWallRight()

SetPhysicsWallRight (iflag)

where:

iflag is an integer value (0 or 1) which switches a wall off (0) or on (1).

When a wall is switched off, physics sprites can move off that edge of the window.

Activity 20.22

Modify *Gravity* so that the sprite can move off screen when the gravity setting pulls it towards the top-right corner.

Test and save your project.

Forces

Sometimes a force can “radiate” from a single point in space. For example, a powerful but small magnet will attract many metal objects towards it. Working on a much larger scale, a planet’s gravitational force will attract passing spacecraft. And in the realms of science fiction (just for the moment), tractor beams and force fields cause similar effects.

CreatePhysicsForce()

FIG-20.52

CreatePhysicsForce()

In AGK we can create this type of force, radiating from a single point on the screen, by using the `CreatePhysicsForce()` statement (see FIG-20.52).

integer CreatePhysicsForce (x , y , fpower , flimit , frange , ifade)

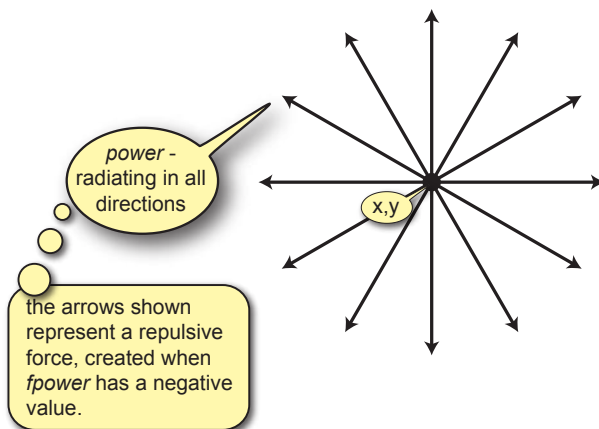
where:

- x,y** are a pair of real values giving the position from which the force is emanating. Use percentage or virtual coordinates as appropriate.
- fpower** is a real number giving the force at one unit's distance from the point (x,y). For other forces we have given *x* and *y* offsets of a vector (see `SetSpritePhysicsForce()` earlier in this chapter) but, since force from a single point radiates in all directions, the figure given here can be thought of as simply the length of such a vector. If a positive value is given, the force will be an attractive one, pulling sprites towards point (x,y); a negative value will create a repulsive force.
- flimit** is a real number giving the maximum force to be exerted when a sprite is less than one unit from the point (x,y). This figure is only relevant when the force exerted varies with distance.
- frange** is a real number giving the range over which the force is effective. Sprites outside that range are not affected by the force. If a negative figure is given, then the range of the force is taken to be unlimited.
- ifade** is an integer value (0 or 1) which determines if the force decreases with the distance between a sprite and point (x,y) (1) or remains fixed (0).

FIG-20.53

A Force Point

The overall idea of this type of force is visualised in FIG-20.53.



The force created is assigned an ID to allow it to be modified by later commands.

The program in FIG-20.54 makes use of the `CreatePhysicsForce()` statement to simulate the moon's pull on a passing spacecraft.

FIG-20.54

Using a Force Point

```
rem *** Physics Force ***  
  
rem *** Load images ***  
LoadImage(1,"Sphere.png")  
LoadImage(2,"Moon.png")  
rem *** Create Sprites ***
```



FIG-20.54

(continued)

Using a Force Point

```

CreateSprite(1,1)
SetSpriteSize(1,3,-1)
SetSpritePosition(1,90,35)
SetSpriteAngle(1,-90)
CreateSprite(2,2)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,42.5,42.5)

rem *** Switch on physics ***
SetSpritePhysicsOn(1,2)
SetSpritePhysicsOn(2,1)

rem *** Modify physics shapes around sprites ***
SetSpriteShapeCircle(1,0,0,1.5)
SetSpriteShapeCircle(2,0,0,7.5)

rem *** No gravity ***
SetPhysicsGravity(0,0)

rem *** No walls ***
SetPhysicsWallBottom(0)
SetPhysicsWallTop(0)
SetPhysicsWallLeft(0)
SetPhysicsWallRight(0)

rem *** Apply force to ship ***
SetSpritePhysicsVelocity(1,-20,0)

rem *** Set moon's force ***
forceid = CreatePhysicsForce(50,50,4,20,15,1)

do
    Sync()
loop

```

➡ The forces used in this program are not meant to represent an accurate model of the moon's gravitational forces.

Activity 20.23

Start a new project called *GravitationalPull* and implement the code given in FIG-20.54. Copy the files *Sphere.png* and *Moon.png* from *AGKDownloads/Chapter20* into the *media* folder.

Test the program.

Modify the force used in `CreatePhysicsForce()` from 4 to -4 and observe how this changes the interaction between the two bodies.

DeletePhysicsForce()

If the force you created using `CreatePhysicsForce()` is only to be a temporary one, you can delete it using the `DeletePhysicsForce()` statement (see FIG-20.55).

FIG-20.55

DeletePhysicsForce()

DeletePhysicsForce (`id`)

where:

id is an integer value giving the ID of the force to be deleted.

SetPhysicsForcePosition()

An existing force can be moved to a different position using the `SetPhysicsForcePosition()` statement (see FIG-20.56).

FIG-20.56

SetPhysicsForcePosition()

`SetPhysicsForcePosition ((id , x , y)`

where:

- id** is an integer value giving the ID of an existing force.
- x,y** are a pair of real numbers giving the new position of the force. The world coordinates should be in percentage or virtual coordinates as appropriate.

SetPhysicsForcePower()

To change the power setting of an existing force, use `SetPhysicsForcePower()` (see FIG-20.57).

FIG-20.57

SetPhysicsForcePower()

`SetPhysicsForcePower ((id , fpower)`

where:

- id** is an integer value giving the ID of an existing force.
- fpower** is a real number giving the new power setting.

SetPhysicsForceRange()

To change the range of an existing force, use `SetPhysicsForceRange()` (see FIG-20.58).

FIG-20.58

SetPhysicsForceRange()

`SetPhysicsForceRange ((id , frange)`

where:

- id** is an integer value giving the ID of an existing force.
- frange** is a real number giving the new range setting.

Summary

- The AGK physics engine assumes that 1% of the screen width (or height) represents one metre irrespective of the actual dimensions of the screen.
- Gravity is set to Earth standard with a pull towards the bottom of the screen.
- Use `SetPhysicsScale()` to set the relationship between screen and real-world distances.
- Use `SetPhysicsGravity()` to set the direction and strength of gravitational pull.
- The edges of the screen have invisible “walls” to prevent objects leaving the screen.

- Use `SetPhysicsWallBottom()`, `SetPhysicsWallTop()`, `SetPhysicsWallLeft()`, and `SetPhysicsWallRight()` to activate/deactivate the physics walls.
- Use `CreatePhysicsForce()` to set a point in space which attracts or repels objects.
- Use `SetPhysicsForcePosition()` to position a point force.
- Use `SetPhysicsForcePower()` to set the strength of a point force.
- Use `SetPhysicsForceRange()` to set the distance over which a point force is effective.
- Use `DeletePhysicsForce()` to delete a point force.

Sprite Physics - 2

Contacts

There are times when we will want to know which sprites are touching. Of course, sprites come into contact with each other when they collide, but contact can also take place when sprites are positioned next to each other. If you want the program logic to react to each of these contacts, then AGK supplies a set of statements to handle this.

FIG-20.59

Sprites in Contact

The code in FIG-20.59 demonstrates a simple multi-collision situation.

```
rem *** Multiple Contacts ***

rem *** Load images ***
LoadImage(1,"Tile.png")
LoadImage(2,"Sphere.png")

rem *** Create sprites and assign physics attributes ***
rem *** Tile sprites ***
for c = 1 to 10
    CreateSprite(c,1)
    SetSpriteSize(c,10,-1)
    SetSpritePosition(c, 30, (c-1)*10)
    SetSpritePhysicsOn(c,2)
    SetSpritePhysicsRestitution(c,0.8)
next c

rem *** Sphere sprite ***
CreateSprite(11,2)
SetSpriteSize(11,6,-1)
SetSpritePosition(11,92,90)
SetSpritePhysicsOn(11,2)
SetSpritePhysicsFriction(11,0)
SetSpritePhysicsVelocity(11,-200,0)
SetSpritePhysicsRestitution(11,0.8)

do
    rem *** Apply force to sphere ***
    SetSpritePhysicsForce(11,95,93,-700,0)
    Sync()
loop
```

Activity 20.24

Create a new project called *MultipleContacts* and implement the code given in FIG-20.59. Copy the files *Tile.png* and *Sphere.png* from *AGKDownloads/Chapter20* to the *media* folder.

Test and save your project.

GetFirstContact()

As a program makes use of the physics engine to calculate the new position of the sprites, it maintains a list of all sprites that come into contact with each other during that moment.

You can determine if any contacts have been made using the `GetFirstContact()` statement (see FIG-20.60).

FIG-20.60

`GetFirstContact()` integer `GetFirstContact` ()

The statement returns 1 if at least one contact has been recorded; if no contacts are recorded, zero is returned.

GetContactSpriteID1() and GetContactSpriteID2()

Once `GetFirstContact()` has been called to check that at least two sprites are in contact, `GetContactSpriteID1()` can be used to discover the ID of the first of the two sprites involved. `GetContactSpriteID2()` will return the ID of the second sprite.

FIG-20.61

`GetContactSpriteID1()` integer `GetContactSpriteID1` ()
`GetContactSpriteID2()` integer `GetContactSpriteID2` ()

The format for each of these two statements is shown in FIG-20.61.

Contact can sometimes be with an object other than a sprite (for example, with a boundary wall). In this case, one of the statements will return zero.

GetContactWorldX() and GetContactWorldY()

The exact position (in world coordinates) at which a contact was made can be discovered using the `GetContactWorldX()` and `GetContactWorldY()` statements.

These statements return coordinates in percentage or virtual coordinates, depending on the measurement system specified in the program.

FIG-20.62

`GetContactWorldX()` float `GetContactWorldX` ()
`GetContactWorldY()` float `GetContactWorldY` ()

The format for each of these statements is given in FIG-20.62.

GetNextContact()

Having accessed the first contact in the list held by the physics engine, you can access the second and subsequent contacts using `GetNextContact()` (see FIG-20.63).

FIG-20.63

`GetNextContact()` integer `GetNextContact` ()

The function will return 1 if another contact is held in the list; zero will be returned if there are no more contacts.

After having determined that this next contact exists, you can then make use of the `GetContactSpriteID1()`, `GetContactSpriteID2()`, `GetContactWorldX()` and `GetContactWorldY()` statements to access details of the contact.

The program in FIG-20.64 is a modification to that given in FIG-20.59. In this version, the tile sprites shrink when in contact with another sprite.

FIG-20.64

Retrieving Contact
Details

```

rem *** Multiple Contacts ***

rem *** Load images ***
LoadImage(1,"Tile.png")
LoadImage(2,"Sphere.png")

rem *** Create sprites and assign physics attributes ***
rem *** Tile sprites ***
For c = 1 to 10
    CreateSprite(c,1)
    SetSpriteSize(c,10,-1)
    SetSpritePosition(c, 30, (c-1)*10)
    SetSpritePhysicsOn(c,2)
    SetSpritePhysicsRestitution(c,0.8)
next c

rem *** Sphere sprite ***
CreateSprite(11,2)
SetSpriteSize(11,6,-1)
SetSpritePosition(11,92,90)
SetSpritePhysicsOn(11,2)
SetSpritePhysicsFriction(11,0)
SetSpritePhysicsVelocity(11,-200,0)
SetSpritePhysicsRestitution(11,0.8)

do
    rem *** Apply force to sphere ***
    SetSpritePhysicsForce(11,95,93,-700,0)
    rem *** IF sprites in contact ***
    if GetFirstContact()= 1
        rem *** Get IDs of sprites in contact ***
        id1 = GetContactSpriteId1()
        id2 = GetContactSpriteId2()
        rem *** IF first ID a tile and second not a wall ***
        if id1 <> 0 and id1 <> 11 and id2 <> 0
            rem *** Reduce size of sprite ***
            SetSpriteSize(id1,GetSpriteWidth(id1)/1.01,
                ↵GetSpriteHeight(id1)/1.01)
        endif
        rem *** IF second ID a tile and first not a wall ***
        if id2 <> 0 and id2 <> 11 and id1 <> 0
            rem *** Reduce size of sprite ***
            SetSpriteSize(id2,GetSpriteWidth(id2)/1.01,
                ↵GetSpriteHeight(id2)/1.01)
        endif
    endif
    Sync()
loop

```

Activity 20.25

Modify *MultipleContacts* to match the code given in FIG-20.64.

Run the program. Does every contact between two sprites result in a size reduction?

Save your project

You should have noticed that not all contacts lead to a reduction in tile size. This was caused by the fact that not all contacts were being processed in each time frame.

We can solve this problem by using `GetNextContact()` to process all the contacts in the list.

Activity 20.26

Modify *MultipleContacts* again, adding a **repeat...until** loop as shown below:

```
if GetFirstContact()
  repeat
    rem *** Get sprite IDs ***
    id1 = GetContactSpriteId1()
    id2 = GetContactSpriteId2()
    rem *** IF first ID a tile and second not a wall ***
    if id1 <> 0 and id1 <> 11 and id2 <> 0
      rem *** Reduce size of sprite ***
      SetSpriteSize(id1, GetSpriteWidth(id1)/1.01,
        ↵ GetSpriteHeight(id1)/1.01)
    endif
    rem *** IF second ID a tile and first not a wall ***
    if id2 <> 0 and id2 <> 11 and id1 <> 0
      rem *** Reduce size of sprite ***
      SetSpriteSize(id2, GetSpriteWidth(id2)/1.01,
        ↵ GetSpriteHeight(id2)/1.01)
    endif
  until GetNextContact() = 0
```

Observe how this changes the nature of the program.

Save your project.

GetSpriteFirstContact()

If you are more interested in whether a specific sprite is involved in a contact, then, rather than use `GetFirstContact()`, you can use `GetSpriteFirstContact()`.

This command allows you to specify a sprite ID and returns 1 if that sprite is in contact with any other elements. The format for `GetSpriteFirstContact()` is shown in FIG-20.65.

FIG-20.65

GetSpriteFirstContact()

integer `GetSpriteFirstContact` ((id))

where:

id is an integer value giving the ID of an existing sprite.

GetSpriteContactSpriteID2()

If `GetSpriteFirstContact()` returns 1, you can discover the ID of the second sprite involved in the contact using `GetSpriteContactSpriteID2()` whose format is shown in FIG-20.66.

FIG-20.66

GetSpriteContactSpriteID2()

integer `GetSpriteContactSpriteID2` (())

The statement returns the ID of the second sprite involved in the contact. If, however, the second element is not a sprite, then zero is returned.

GetSpriteContactWorldX() and GetSpriteContactWorldY()

The point of contact between the two elements can be determined using `GetSpriteContactWorldX()` and `GetSpriteContactWorldY()`. The format for each of these statements is shown in FIG-20.67.

FIG-20.67

<code>GetSpriteContactWorldX()</code>	float	<code>GetSpriteContactWorldX</code>	()
<code>GetSpriteContactWorldY()</code>	float	<code>GetSpriteContactWorldY</code>	()

If the screen has been scrolled or zoomed, these world coordinates will not match screen coordinates.

GetSpriteNextContact()

If the sprite you specified when calling `GetSpriteFirstContact()` was involved in other contacts at the same time as the first, then `GetSpriteNextContact()` will return 1 when called, otherwise zero is returned.

FIG-20.68

<code>GetSpriteNextContact()</code>	integer	<code>GetSpriteNextContact</code>	()
-------------------------------------	---------	-----------------------------------	---	---

The format for `GetSpriteNextContact()` is shown in FIG-20.68.

The next program (see FIG-20.69) is a variation on the last project. This time only tiles that come into contact with the sphere are reduced in size.

FIG-20.69

Using Sprite Contacts

```
rem *** Multiple Contacts ***

rem *** Load images ***
LoadImage(1,"Tile.png")
LoadImage(2,"Sphere.png")

rem *** Create sprites and assign physics attributes ***

rem *** Tile sprites ***
For c = 1 to 10
  CreateSprite(c,1)
  SetSpriteSize(c,10,-1)
  SetSpritePosition(c, 30, (c-1)*10)
  SetSpritePhysicsOn(c,2)
  SetSpritePhysicsRestitution(c,0.8)
next c

rem *** Sphere sprite ***
CreateSprite(11,2)
SetSpriteSize(11,6,-1)
SetSpritePosition(11,92,90)
SetSpritePhysicsOn(11,2)
SetSpritePhysicsFriction(11,0)
SetSpritePhysicsVelocity(11,-200,0)
SetSpritePhysicsRestitution(11,0.8)
```



FIG-20.69

(continued)

Using Sprite Contacts

```

do
    rem *** Apply force to sphere ***
    SetSpritePhysicsForce(11,95,93,-700,0)
    rem *** IF sphere in contact ***
    if GetSpriteFirstContact(11)
        rem *** REPEAT for all contacts ***
        repeat
            rem *** Get contact ID ***
            id2 = GetSpriteContactSpriteId2()
            rem *** IF it is a tile THEN ***
            if id2 <> 0
                rem *** Reduce the tile's size ***
                SetSpriteSize(id2,GetSpriteWidth(id2)/1.1,
                    ↳GetSpriteHeight(id2)/1.1)
            endif
        until GetSpriteNextContact() = 0
    endif
    Sync()
loop

```

Activity 20.27

Modify *MultipleContacts* to match the code given in FIG-20.69.

Run and save the program.

Physics Groups and Categories

Back in Chapter 17 we saw how sprite groups could be used to ensure only hits on a specific group of sprites could be detected. We can make use of that same grouping statement (`SetSpriteGroup()`) to affect sprite collisions. Normally, sprites collide irrespective of the group to which they belong, but if we assign a negative value to the group identity, then sprites in that group will not collide.

The program in FIG-20.70 creates 10 apples and 10 oranges which fly about the screen. All apple sprites belong to group -1 and will not collide with each other; oranges belong to group -2 and, again, do not collide. However, an apple will collide with an orange since they belong to different groups.

FIG-20.70

How Groupings Affect Collisions

```

rem *** Physics Groups and Collisions ***

rem *** Set screen background ***
SetClearColor(25,100,20)
Sync()

rem *** Load images ***
LoadImage(1,"Apple.png")
LoadImage(2,"Orange.png")

rem *** Create 10 apple sprites ***
CreateSprite(1,1)
SetSpriteSize(1,6,-1)
SetSpritePosition(1,Random(0,92),Random(0,92))

```



FIG-20.70

(continued)

How Groupings Affect Collisions

```

for c = 2 to 10
    CloneSprite(c,1)
    SetSpritePosition(c,Random(0,92),Random(0,92))
next c

rem *** Create 10 orange sprites ***
CreateSprite(11,2)
SetSpriteSize(11,6,-1)
SetSpritePosition(11,Random(0,92),Random(0,92))
for c = 12 to 20
    CloneSprite(c,11)
    SetSpritePosition(c,Random(0,92),Random(0,92))
next c
rem *** Group apples ***
for c = 1 to 10
    SetSpriteGroup(c,-1)
next c

rem *** Group oranges ***
for c = 11 to 20
    SetSpriteGroup(c,-2)
next c

rem *** Switch on physics ***
for c = 1 to 20
    SetSpriteShape(c,1)
    SetSpritePhysicsOn(c,2)
    angle = Random(0,359)
    SetSpritePhysicsVelocity(c,Cos(angle)*50,Sin(angle)*50)
    SetSpritePhysicsRestitution(c,1)
next c

rem *** No gravity ***
SetPhysicsGravity(0,0)

do
    Sync()
loop

```

Activity 20.28

Start a new project called *PhysicsGroups* and implement the code in FIG-20.70. Copy *Apple.png* and *Orange.png* to the *media* folder.

Test and save the program.

When two sprites belong to the same group, they will always collide if that group number is positive (greater than zero) and always slide past each other without colliding if that group number is negative.

The physics engine can also make use of a sprite's categories to determine if a collision is to take place. This happens when sprites belong to different groups or to group zero (the default group).

Back in Chapter 17 we used the `SetSpriteCategoryBits()` and `SetSpriteCategoryBit()` statements to set the categories to which a sprite belonged. The value set with these instructions affect the collision characteristics of sprites.

To see how this works, let's revisit the apples and oranges example and introduce a new type - green apples.

All apples (red and green) will be in group -1 and oranges in group -2. However, green apples will belong to category 4, red apples to category 3, and oranges to category 2 as shown in FIG-20.71.

FIG-20.71

Sprite Categories

	Category Bits																Green Apples Red Apples Oranges Other Sprites			
	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1				
Oranges	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0				
Red Apples	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0				
Green Apples	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0				
Other Sprites	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1				

SetSpriteCollideBits()

To decide which categories will not collide, we need to build a 16-bit mask value which is then ANDed with the category settings. Any bits set to 1 in the final result will collide when the sprites belong to different groups.

FIG-20.72

This mask value is constructed using `SetSpriteCollideBits()` (see FIG-20.72).

SetSpriteCollideBits()

`SetSpriteCollideBits (id , imask)`

where

- id** is an integer value giving the ID of the sprite whose collision detection categories are to be set.
- imask** is an integer value giving the mask to be ANDed with the categories of any other sprite that comes into contact with the specified sprite.

For example, let's say we want oranges to collide with red apples but not green ones, then, for an orange sprite, *c*, we would use the line:

```
SetSpriteCollideBits(c,%1001)
```

Now you may be forgiven for thinking that there are no other sprites in the demonstration other than the oranges and apples, but, in fact, the edge "walls" are hidden sprites and if you don't allow the oranges to collide with those "other sprites", the oranges will disappear off the edge of the screen.

The program in FIG-20.73 is a variation on the earlier Apples and Oranges project with 5 red and 5 green apples. The oranges will only collide with the red apples.

FIG-20.73

Using Sprite Contacts to
Control Collisions

```
rem *** Sprite Categories and Collisions ***

rem *** Set background colour ***
SetClearColor(150,150,170)
Sync()

rem *** Load images ***
LoadImage(1,"Apple.png")
LoadImage(2,"Orange.png")
LoadImage(3,"GreenApple.png")

rem *** Create 10 apple sprites ***
CreateSprite(1,1)
SetSpriteSize(1,6,-1)
SetSpritePosition(1,Random(0,92),Random(0,92))
rem *** Red apples ***
for c = 2 to 10
    CloneSprite(c,1)
    SetSpritePosition(c,Random(0,92),Random(0,92))
next c
rem *** Change 5 apples to green ***
for c = 6 to 10
    SetSpriteImage(c,3)
next c

rem *** Create 10 orange sprites ***
CreateSprite(11,2)
SetSpriteSize(11,6,-1)
SetSpritePosition(11,Random(0,92),Random(0,92))
for c = 12 to 20
    CloneSprite(c,11)
    SetSpritePosition(c,Random(0,92),Random(0,92))
next c

rem *** Group apples ***
for c = 1 to 10
    SetSpriteGroup(c,-1)
next c
rem *** Set the category for red apples ***
for c = 1 to 5
    SetSpriteCategoryBits(c,%100)
next c

rem *** Set the category for green apples ***
for c = 6 to 10
    SetSpriteCategoryBits(c,%1000)
next c

rem *** Group and categorise oranges ***
for c = 11 to 20
    SetSpriteGroup(c,-2)
    rem *** Set the category for oranges ***
    SetSpriteCategoryBits(c,%10)
    rem *** Set the categories with which oranges can collide ***
    SetSpriteCollideBits(c,%101)
next c

rem *** Switch on physics ***
for c = 1 to 20
    SetSpriteShape(c,1)
    SetSpritePhysicsOn(c,2)
```



FIG-20.73

(continued)

Using Sprite Contacts to
Control Collisions

```
angle = Random(0,359)
SetSpritePhysicsVelocity(c,Cos(angle)*50,Sin(angle)*50)
SetSpritePhysicsRestitution(c,1)
next c

rem *** No gravity ***
SetPhysicsGravity(0,0)

do
    Sync()
loop
```

Activity 20.29

Modify *PhysicsGroups* to match the code in FIG-20.73. Copy *GreenApple.png* to the *media* folder. Test and save the program.

SetSpriteCollideBit()

You can set individual bits within the collision mask using `SetSpriteCollideBit()` (see FIG-20.74).

FIG-20.74

SetSpriteCollideBit()

`SetSpriteCollideBit` (`id` , `index` , `iflag`)

where

- | | |
|--------------|--|
| id | is an integer value giving the ID of the sprite. |
| index | is an integer value (1 to 16) giving the mask bit whose value is to be set. |
| iflag | is an integer value (0 or 1) giving the mask bit setting.
(0: don't collide; 1: collide). |

For example, we could reset bit 1 of sprite *c*'s mask using the line:

```
SetSpriteCategoryBit(c,1,0)
```

Activity 20.30

Modify *PhysicsGroups* so that oranges collide with green apples after the program has been running for 10 seconds. Test and save the program.

CreateDummySprite()

Although there may be no immediately obvious reason for having a sprite which is not capable of appearing on the screen, they can, in fact, be useful in some situations where the physics engine is being used.

You can create such a “dummy” sprite using the `CreateDummySprite()` statement (see FIG-20.75).

CreateDummySprite()

where

id is an integer value giving the ID to be assigned to the dummy sprite.

You can add bounding shapes to a dummy sprite and detect any collisions with other physics-enabled sprites.

Physics Ray Casting

We have already looked at the principles of ray casting in Chapter 17. However, when dealing with physics-enabled sprites, we do not use the `SpriteRayCast()` statement to initiate a ray cast.

PhysicsRayCast()

To initiate a ray cast which detects physics-activated elements such as sprites and edge walls, use the `PhysicsRayCast()` statement (see FIG-20.76).

FIG-20.76

PhysicsRayCast()

integer PhysicsRayCast (x1 , y1 , x2 , y2)

where:

x1,y1 are a pair of real values giving the coordinates of the starting point of the ray cast.

x2,y2 are a pair of real values giving the coordinates of the finishing point of the ray cast.

If the ray intersects one or more physics elements, then `PhysicsRayCast()` returns 1, otherwise zero is returned. The details of the first element encountered by the ray cast are stored and can be accessed by the other ray cast commands covered in Chapter 17 such as `GetRayCastSpriteID()`, `GetRayCastX()`, `GetRayCastY()`, etc.

If the ray cast starts from a position within a sprite, that sprite is ignored in the calculations performed.

The program in FIG-20.77 sets up a rotating cannon which uses ray casting to detect when it is in line with a cherry sprite. When the cherry is detected, the cannon fires a missile at its target.

FIG-20.77

Using a Physics Ray Cast

```
rem *** Physics Ray Casting ***

rem *** Global variables ***
global angle = 0 //Turret angle of rotation
global count = 0 //Missile count
global fired = 0 //Missile fired (0:no; 1:Yes)

rem *** Main program logic ***
LoadImages()
SetUpScreen()
SetUpPhysics()
```



FIG-20.77

(continued)

Using a Physics Ray Cast

```

do
    Play()
    Sync()
loop

rem *** Functions ***

rem *** Load Images ***
function LoadImages()
    LoadImage(1,"Turret2.png")
    LoadImage(2,"Cherry.png")
    LoadImage(3,"Shell.png")
endfunction

rem *** Set up screen layout ***
function SetUpScreen()
    rem ** Set screen colour ***
    SetClearColor(120,120,120)
    Sync()
    rem *** Turret ***
    CreateSprite(1,1)
    SetSpriteSize(1,15,-1)
    SetSpritePosition(1,42.5,42.5)
    SetSpriteOffset(1,5,GetSpriteHeight(1)/2.0)
    rem *** Cherry ***
    CreateSprite(2,2)
    SetSpriteSize(2,7,-1)
    SetSpritePosition(2,45,10)
    rem *** Projectile ***
    CreateSprite(3,3)
    SetSpriteSize(3,2,-1)
    SetSpritePositionByOffset(3,GetSpriteXByOffset(1),
    ↳GetSpriteYByOffset(1))
    SetSpriteDepth(3,11)
endfunction

rem *** Set up Physics ***
function SetUpPhysics()
    SetSpritePhysicsOn(2,2)
    SetSpritePhysicsRestitution(2,0.8)
    SetSpritePhysicsOn(3,2)
    SetSpritePhysicsIsBullet(3,1)
    rem *** Set gravity off ***
    SetPhysicsGravity(0,0)
endfunction

function Play()
    rem *** Turn turret ***
    SetSpriteAngle(1,angle)
    rem *** Calculate coordinates of end of barrel ***
    barrelx = GetSpriteXByOffset(1) +10*cos(angle)
    barrely = GetSpriteYByOffset(1) + 10*sin(angle)
    rem *** Ray cast out from the barrel ***
    hit = PhysicsRaycast(barrelx,barrely,(100*cos(angle))+
    ↳barrelx,100*(Sin(angle))+barrely)

```



FIG-20.77

(continued)

Using a Physics Ray Cast

```

rem *** If a hit, get sprite ID and add to count***
if hit = 1
    id = GetRayCastSpriteId()
    if id = 2 and fired = 0
        count = count + 1
        if count >= 4
            SetSpriteAngle(3,angle)
            SetSpritePhysicsOn(3,2)
            SetSpritePositionByOffset(3,barrelx,barrely)
            SetSpritePhysicsImpulse(3,barrelx,barrely,
                ↵20*Cos(angle),20*Sin(angle))
            fired = 1
        endif
    endif
endif
endif
rem *** If shell off edge of screen, delete it ***
shellx = GetSpriteX(3)
shelly = GetSpriteY(3)
if shellx < 4 or shellx > 96 or shelly < 4 or shelly > 96
    ↵and fired = 1
        SetSpritePhysicsOff(3)
        SetSpritePositionByOffset(3,GetSpriteXByOffset(1),
            ↵GetSpriteYByOffset(1))
        count = 0
        fired = 0
    endif
endif
Sync()
rem *** Change angle of turret rotation ***
angle = (angle + 2) mod 360
endfunction

```

Activity 20.31

Start a new project called *PhysicsRayCasting* and implement the code given in FIG-20.77. Copy the necessary image files to the *media* folder.

Test and save the program.

PhysicsRayCastGroup()

If you wish a ray cast to detect only sprites that are within a specific group, then you can create a ray cast using the **PhysicsRayCastGroup()** statement (see FIG-20.78).

FIG-20.78 PhysicsRayCastGroup()

integer **PhysicsRayCastGroup** ((**igrp** , **x1** , **y1** , **x2** , **y2**))

where

igrp is an integer value giving the group to be checked by the ray cast.

x1,y1 are a pair of real values giving the coordinates of the starting point of the ray cast.

x2,y2 are a pair of real values giving the coordinates of the finishing point of the ray cast.

If the ray cast hits a sprite belonging to the specified group, the ID of that sprite is returned by the function. If no sprite in that group is hit, zero is returned.

For example, we could check if a ray from the centre of the screen to the top-right corner hits any sprite in group 5 with the line:

```
PhysicsRayCastGroup(5,50,50,100,0)
```

PhysicsRayCastCategory()

To use a ray cast which checks only for sprites in specified categories, use `PhysicsRayCastCategory()` (see FIG-20.79).

FIG-20.79 `PhysicsRayCastCategory()`

integer `PhysicsRayCastCategory` ((`icat` , `x1` , `y1` , `x2` , `y2`))

where

- | | |
|--------------|--|
| icat | is an integer value giving the categories to be checked by the ray cast. |
| x1,y1 | are a pair of real values giving the coordinates of the starting point of the ray cast. |
| x2,y2 | are a pair of real values giving the coordinates of the finishing point of the ray cast. |

For example, we could check if a ray from the top-left corner to the bottom-right corner encounters a sprite in the first four categories using the line:

```
PhysicsRayCastCategory(%1111,0,0,100,100)
```

Summary

- Use `GetFirstContact()` to discover the first in the list of contacts between sprites made during the current time period.
- Use `GetNextContact()` to access each subsequent contact.
- Use `GetSpriteContactID1()` and `GetSpriteContactID2()` to find the IDs of the sprites involved in the current collision.
- Use `GetContactWorldX()` and `GetContactWorldY()` to find the point of collision for the current contact.
- Use `GetSpriteFirstContact()` to access the first in the list of contacts made with a specific sprite.
- Use `GetSpriteNextContact()` to access subsequent contacts for a given sprite.
- Use `GetSpriteContactID2()` to discover the ID of the second sprite involved in the current collision.
- Use `GetSpriteContactWorldX()` and `GetSpriteWorldContactY()` to determine the point of contact between the specified sprite and second sprite.
- Physics sprites can be assigned to groups and categories in the same way as regular sprites.
- Use `SetSpriteCollideBits()` to specify which categories of sprite will be

included when checking for a collision.

- Use `SetSpriteCollideBit()` to set a specific bit in the collision category mask.
- Use `CreateDummySprite()` to create a dummy sprite.
- Use `PhysicsRayCast()` to create a ray cast which only detects physics-enabled sprites.
- Use `SetPhysicsRayCastGroup()` to specify which sprite group can be detected when ray casting.
- Use `SetPhysicsRayCastCategory()` to specify which categories can be detected by a ray cast.

Introduction

Joins allow us to join two physics-activated sprites so that they automatically interact with each other in some specific way.

Joins attempt to simulate real-world situations where two elements have a fixed relationship to each other. For example, we might wish to simulate one car being towed by another, how a piston moves, how a human head moves in relation to the torso, or how the gears of a mechanical devise interact.

Exactly how the linked sprites interact depends on the joint option selected.

Joint Statements

CreateWeldJoint()

Perhaps the simplest joint is the weld joint which fixes the position of two sprites relative to each other. We may think of the two sprites as being magnetically attracted to each other, unable to move apart or to rotate in relation to each other. Under normal forces, the sprites will stay together and cannot rotate relative to one another, but if a strong enough force is applied, they can be forced apart - at least temporarily.

FIG-20.80

A weld joint is achieved using the `CreateWeldJoint()` statement whose format is given in FIG-20.80.

CreateWeldJoint()

Format 1

`CreateWeldJoint` ((`id` , `sprId1` , `sprId2` , `x` , `y` , `icold`))

Format 2

integer `CreateWeldJoint` ((`sprId1` , `sprId2` , `x` , `y` , `icold`))

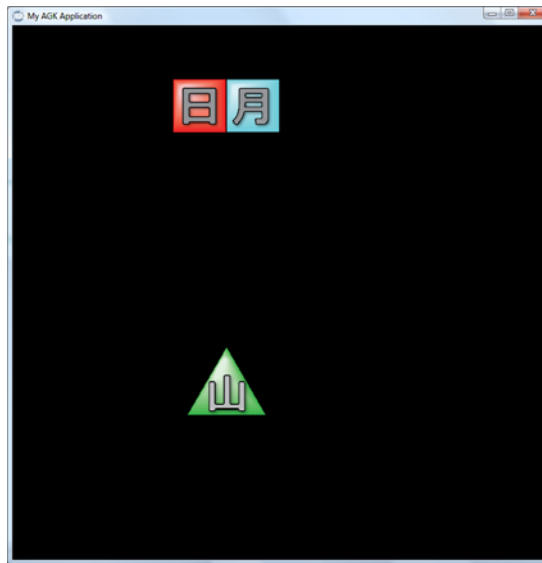
where

- | | |
|---------------|---|
| id | is an integer value giving the ID to be assigned to the joint. |
| sprId1 | is an integer value giving the ID of the first sprite involved in the joint. |
| sprId2 | is an integer value giving the ID of the second sprite involved in the joint. |
| x,y | are real values giving the coordinates of the anchor point of the joint. |
| icold | is an integer value specifying whether the sprites may, under certain circumstances, collide. |

If the concept of joints is new to you, they may take a little getting used to. So to try and give a better idea of how they operate, we will create a program containing the setup shown in FIG-20.81.

FIG-20.81

Sprites in a Weld Joint



The top two sprites (*day* and *month*) have been placed side-by-side and joined using a weld joint. Gravity will cause these two dynamic sprites to fall onto the third, static sprite (*mountain*).

FIG-20.82

Using a Weld Joint

The code for the demonstration is given FIG-20.82.

```
rem *** A Weld Joint ***

rem *** Load Images ***
LoadImage(1,"Day.png")
LoadImage(2,"Month.png")
LoadImage(3,"Mountain.png")

rem *** Create sprites ***
CreateSprite(1,1)
SetSpriteSize(1,10,10)
SetSpritePosition(1,30,10)
CreateSprite(2,2)
SetSpriteSize(2,10,10)
SetSpritePosition(2,40,10)
CreateSprite(3,3)
SetSpriteSize(3,15,-1)
SetSpritePosition(3,32.5,60)
rem *** Create polygon bounding area for triangle ***
SetSpriteShape(3,3)

rem *** Switch on physics ***
SetSpritePhysicsOn(1,2)    //Dynamic
SetSpritePhysicsOn(2,2)    //Dynamic
SetSpritePhysicsOn(3,1)    //Static

rem *** Create Weld Joint ***
CreateWeldJoint(1,1,2,40,15,1)

rem *** Watch results ***
do
    Sync()
loop
```

Note that the mountain sprite's bounding area is changed from the default rectangle to a more accurate polygon using the line:

```
SetSpriteShape(3,3)
```

Also, since the mountain sprite is static, it will not be effected by the gravitational pull.

Activity 20.32

Start a new project called *Joints01* and implement the code given in FIG-20.82.

Copy the files *Day.png*, *Month.png* and *Mountain.png* into the project's *media* folder.

Test your project, observing how the two joined sprites behave.

Save your project.

We can gain a greater insight into what is going on by using the physics debug option. This will not only show the bounding areas of the sprites but also the joint between the two sprites.

Activity 20.33

Modify *Joints01* by adding the lines

```
rem *** Switch on physics debug ***  
SetPhysicsDebugOn()
```

after all three sprites have had the physics engine activated.

Test your project, observing the line showing the joint between the sprites.

Save your project.

By using enough force, we can get the sprites to separate - at least temporarily.

Activity 20.34

Modify *Joints01* by adding the lines

```
rem *** Set gravity ***  
SetPhysicsGravity(0,120)
```

after physics debug has been activated.

Test your project, observing the line showing the joint between the sprites.

Change the gravitational force downwards from 120 to 1200 and see how this changes the result.

Save your project.

Although it will not reflect real-life situations, it is possible to start with the two

sprites some distance apart. The weld joint will attempt to maintain this gap.

Activity 20.35

Modify *Joints01* by changing the starting position of sprite 2 from (40,10) to (60,10).

Test your project, observing the line showing the joint between the sprites.

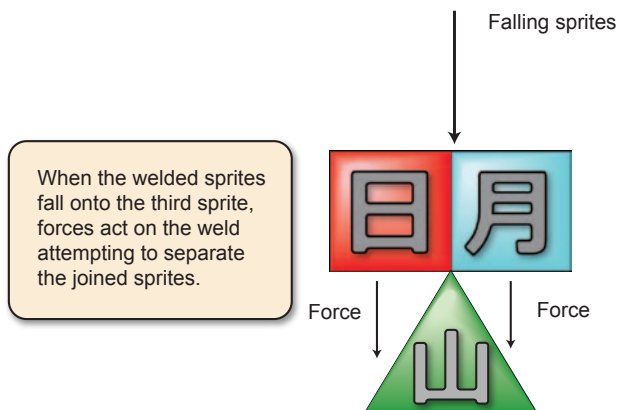
Save your project.

GetJointReactionForceX() and GetJointReactionForceY()

As we saw when running *Joints01*, a joint can be subjected to forces which attempt to break it apart (see FIG-20.83).

FIG-20.83

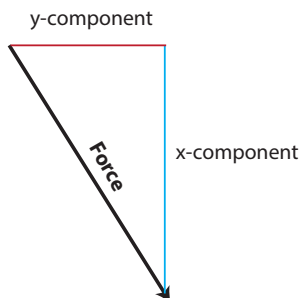
Forces on a Joint



When the joined rectangles hit the triangular sprite, they are temporarily separated by those forces. Like a velocity, a force can be described in terms of its *x* and *y* components (see FIG-20.84).

FIG-20.84

Force Components



You can determine the forces acting on a joint using the `GetJointReactionForceX()` and `GetJointReactionForceY()` statements which return the *x* and *y* components of that force (see FIG-20.85).

FIG-20.85

`GetJointReactionForceX()`

float `GetJointReactionForceX` ((id))

`GetJointReactionForceY()`

float `GetJointReactionForceY` ((id))

where

id

is an integer value giving the ID of the joint.

Activity 20.36

Modify *Joints01* by returning the starting position of sprite 2 to (40,10).

Add the lines

```
x# = GetJointReactionForceX(1)
y# = GetJointReactionForceY(1)
Print("Force on joint  X: "+Str(x#)+" Y: "+Str(y#))
```

between the

```
do
```

and

```
Sync()
```

statements.

Test your project, observing the values displayed as the blocks fall.

It is difficult to see how strong the force on the joint becomes since it changes so quickly. It might help if only the strongest force is displayed.

Modify the `do...loop` code to read

```
do
    x# = GetJointReactionForceX(1)
    y# = GetJointReactionForceY(1)
    force# = Sqrt(x#*x# + y#*y#)
    if force# > largestforce#
        largestforce# = force#
        largestx# = x#
        largesty# = y#
    endif
    Print("Largest force on joint  x: "+Str(largestx#)+
        " Y: "+Str(largesty#)) Sync()
loop
```

Run your program several times and watch for when the largest force is applied to the joint.

Save your project.

As you saw from the results of Activity 20.36, the largest force can often occur when the blocks come to rest. The weight of one block is pushing down on the joint. However, this force is not attempting to break the joint; in fact, it only helps strengthen the bond between the two blocks.

DeleteJoint()

FIG-20.86

A joint can be deleted using the `DeleteJoint()` statement (see FIG-20.86).

DeleteJoint()

DeleteJoint ((id))

where

id

is an integer value giving the ID of the joint to be deleted.

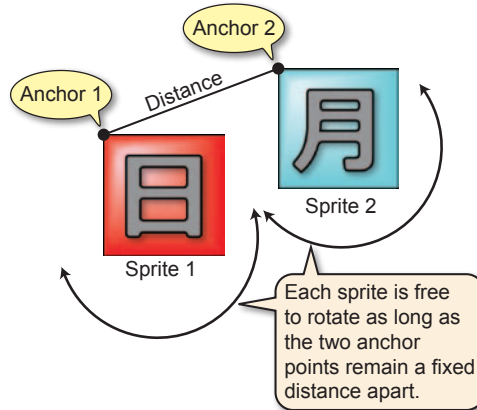
CreateDistanceJoint()

The distance joint is similar to the weld joint but, in this case, the joint ensures that the anchor points on the two sprites stay a fixed distance apart (which is similar to a weld joint) but the sprites may rotate relative to each other.

Once the sprites are in their initial position on the screen, the distance joint should be created. This involves creating two anchor points (one on each sprite) and the distance joint will ensure these two points remain a fixed distance apart. The idea of the joint is expressed visually in FIG-20.87.

FIG-20.87

Characteristics of a Distance Joint



A distance joint is created using the `CreateDistanceJoint()` statement (see FIG-20.88).

FIG-20.88 `CreateDistanceJoint()`

Format 1

```
CreateDistanceJoint ( ( id , sprId1 , sprId2 , x1 , y1 , x2 , y2 , icold ) )
```

Format 2

```
integer CreateDistanceJoint ( ( sprId1 , sprId2 , x1 , y1 , x2 , y2 , icold ) )
```

where

- | | |
|---------------|--|
| id | is an integer value giving the ID to be assigned to the joint. |
| sprId1 | is an integer value giving the ID of the first sprite involved in the joint. |
| sprId2 | is an integer value giving the ID of the second sprite involved in the joint. |
| x1,y1 | are real values giving the coordinates of the anchor point for the first sprite. Normally, this will be a point within the body of the sprite. |
| x2,y2 | are real values giving the coordinates of the anchor point for the second sprite (normally, within the sprite's body). |
| icold | is an integer value specifying whether the sprites may, under certain circumstances, collide. |

Once set up, the physics will attempt to maintain the distance between anchor points 1 and 2. The anchor points can be at any point within each sprite. Often the anchor points will be at the centre of each sprite, but you are free to position the anchor points anywhere - even outside the area of the sprite.

The program in FIG-20.89 demonstrates the effect of a distance joint.

FIG-20.89

Using a Distance Joint

```
rem *** A Distance Joint ***

rem *** Load Images ***
LoadImage(1,"Day.png")
LoadImage(2,"Month.png")
LoadImage(3,"Mountain.png")

rem *** Create sprites ***
CreateSprite(1,1)
SetSpriteSize(1,10,10)
SetSpritePosition(1,30,10)
CreateSprite(2,2)
SetSpriteSize(2,10,10)
SetSpritePosition(2,60,10)
CreateSprite(3,3)
SetSpriteSize(3,15,-1)
SetSpritePosition(3,32.5,60)
SetSpriteShape(3,3)

rem *** Switch on physics ***
SetSpritePhysicsOn(1,2)
SetSpritePhysicsOn(2,2)
SetSpritePhysicsOn(3,1)

rem *** Set gravity ***
SetPhysicsGravity(0,120)

rem *** Switch on physics debug ***
SetPhysicsDebugOn()

rem *** Create Distant Joint ***
CreateDistanceJoint(1,1,2,GetSpriteX(1),GetSpriteY(1),
↳GetSpriteX(2),GetSpriteY(2),1)

rem *** Run simulation ***
do
    Sync()
loop
```

Activity 20.37

Create a new project called *JointDistance* and copy the three images required into the *media* folder. Implement and test the code given in FIG-20.89.

Modify the code so that the centre of each sprite is given as an anchor point for the distance joint.

Test and save your project.

CreateMouseJoint()

The mouse joint is used to force the anchor point within a sprite to try to reach a specific position in 2D space. Often this point will be the position of the mouse pointer - hence the name of the joint. A mouse joint is created using the `CreateMouseJoint()` statement (see FIG-20.90).

FIG-20.90

CreateMouseJoint()

Format 1

`CreateMouseJoint` ((`id` , `sprId1` , `x` , `y` , `force`))

Format 2

integer `CreateMouseJoint` ((`sprId1` , `x` , `y` , `force`))

where

- id** is an integer value giving the ID to be assigned to the joint.
- sprId1** is an integer value giving the ID of the sprite involved in the joint.
- x,y** are real values giving the coordinates of the anchor point of the sprite.
- force** is an real value specifying the maximum force that can be applied to move the sprite towards the target point.

SetJointMouseTarget()

Creating a mouse joint is only half the job needed. The second stage is to specify the target point. It is this position that the anchor point of the mouse joint sprite will attempt to reach.

The target point is defined using the `SetJointMouseTarget()` statement (see FIG-20.91).

FIG-20.91

SetJointMouseTarget()

`SetJointMouseTarget` ((`id` , `x` , `y`))

where

- id** is an integer value giving the ID of the existing mouse joint.
- x,y** are real values giving the world coordinates of the target position for the mouse joint anchor point.

Creating a target point has no visual effect on the screen.

The program in FIG-20.92 creates a ball (placed initially near the top-left corner) and a target point near the bottom-right corner. As the ball falls under the pull of gravity it rolls towards the target point.

FIG-20.92

Using a Mouse Joint and Target

```
rem *** A Mouse Joint ***

rem *** Load Image ***
LoadImage(1,"Shape02.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,7,-1)
SetSpritePosition(1,10,10)

rem *** Switch on physics ***
SetSpritePhysicsOn(1,2)
rem *** Set circular bounding box ***
SetSpriteShape(1,1)

rem *** Create mouse joint ***
CreateMouseJoint(1,1,GetSpriteXByOffset(1),
↳GetSpriteYByOffset(1),2)

rem *** Create target ***
SetJointMouseTarget(1,80,96.5)

rem *** Run simulation ***
do
    Sync()
loop
```

Activity 20.38

Start a new project called *JointMouse* and implement the code in FIG-20.92. Copy *Shape02.png* to the *media* folder.

Run the program and observe the effect produced. Save your project.

It only takes a little more code to set the target point using the mouse, thereby making the name of the joint a little more appropriate.

Activity 20.39

Modify *JointMouse* by adding the lines

```
if GetPointerPressed()=1
    SetJointMouseTarget(1,GetPointerX(),GetPointerY())
endif
```

at the start of the `do..loop` structure.

Test your program by clicking on various positions within the app window.

CreateRevoluteJoint()

Perhaps the most widely used joint is the revolute joint. A revolute joint allows you to specify a common point about which one sprite can revolve in relation to another. For example, we could use this type of joint to simulate the blades of a windmill revolving about the windmill tower, wheels turning on a car, a door swinging about

a door frame, or your head moving in relation to your torso. Of course, in the last two cases, only a limited amount of movement can be achieved.

A revolute joint is produced using the `CreateRevoluteJoint()` statement (see FIG-20.93).

FIG-20.93 `CreateRevoluteJoint()`

Format 1

`CreateRevoluteJoint` (`id` , `sprId1` , `sprId2` , `x` , `y` , `icold`)

Format 2

integer `CreateRevoluteJoint` (`sprId1` , `sprId2` , `x` , `y` , `icold`)

where

- id** is an integer value giving the ID to be assigned to the joint.
- sprId1** is an integer value giving the ID of the first sprite involved in the joint.
- sprId2** is an integer value giving the ID of the second sprite involved in the joint.
- x,y** are real values giving the coordinates of the anchor point about which the sprites will revolve.
- icold** is an integer value specifying whether the sprites may, under certain circumstances, collide.

FIG-20.94 shows a seesaw simulation setup with a seating bar balanced on a central pivot. In this situation, the position of the pivot is fixed and the bar can rotate about the apex of the pivot.

FIG-20.94

A Typical Pivot

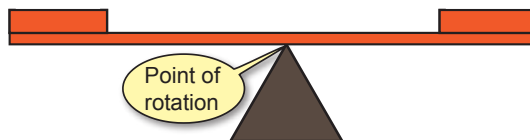


FIG-20.95

Using a Revolute Joint

The program in FIG-20.95 uses this setup to demonstrate one use of a revolute joint.

```
rem *** A Revolute Joint Pivot ***

rem *** Load Image ***
LoadImage(1,"Pivot.png")
LoadImage(2,"Bar.png")

rem *** Create sprites ***
rem *** Pivot ***
CreateSprite(1,1)
SetSpriteSize(1,8,-1)
SetSpritePosition(1,46,69)
rem *** Balance ***
```



FIG-20.95

(continued)

Using a Pivot Joint

```

CreateSprite(2,2)
SetSpriteSize(2,40,-1)
SetSpritePosition(2,30,67)
rem *** Set background colour ***
ClearColor(120,120,120)
Sync()
rem *** Switch on physics ***
SetSpritePhysicsOn(1,1)
SetSpritePhysicsOn(2,2)
rem *** Change pivot bounding box to a polygon ***
SetSpriteShape(1,3)
rem *** Create joint ***
CreateRevoluteJoint(1,1,2,GetSpriteXByOffset(1),GetSpriteY(1),1)
rem *** Set gravity ***
SetPhysicsGravity(0,32)
rem *** Run simulation ***
do
    Sync()
loop

```

► For this project, set the window size to 768 x 1024.

Activity 20.40

Start a new project called *JointRevolute01* and, from *AGKDownloads/Chapter20*, copy the files *Pivot.png* and *Bar.png* to the project's *media* folder.

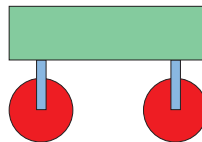
Implement the code given in FIG-20.95. Why does the bar not move?

Change the position of the bar sprite from (30,67) to (31,67). How does this affect the simulation?

Modify the `CreateRevoluteJoint()` statement so that the sprites do not collide. How does this affect the result? Save your project.

FIG-20.96

A Model Using Revolute Joints



In this next example, two wheels are used to create a moving vehicle (see FIG-20.96).

The wheels are attached to the vertical struts using revolute joints, while the struts are attached to the body of the vehicle using weld joints.

FIG-20.97

Using a Pivot Joint

FIG-20.97 shows the code used to create the setup.

```

rem *** A Revolute Joint Wheels ***
rem *** Load Image ***
LoadImage(1,"Wheel.png")
LoadImage(2,"Strut.png")
LoadImage(3,"Container.png")
rem *** Create sprites ***
rem *** Wheel ***
CreateSprite(1,1)
SetSpriteSize(1,8,-1)
SetSpritePosition(1,5,90)

```



FIG-20.97

(continued)

Using a Pivot Joint

```

rem *** Strut ***
CreateSprite(2,2)
SetSpriteSize(2,2,-1)
SetSpritePosition(2,8,86)

rem *** Container ***
CreateSprite(3,3)
SetSpriteSize(3,30,-1)
SetSpritePosition(3,5,80)
SetSpriteDepth(3,9)

rem *** Create second wheel ***
CloneSprite(4,1)
SetSpritePosition(4,27,90)

rem *** Create second strut ***
CloneSprite(5,2)
SetSpritePosition(5,30,86)

rem *** Set background colour ***
SetClearColor(120,120,120)

rem *** Switch on physics ***
SetSpritePhysicsOn(1,2) //Dynamic
SetSpritePhysicsOn(2,2) //Dynamic
SetSpritePhysicsOn(3,2) //Dynamic
SetSpritePhysicsOn(4,2) //Dynamic
SetSpritePhysicsOn(5,2) //Dynamic
SetSpriteShape(1,1) //Circular bounding box
SetSpriteShape(4,1) //Circular bonding box

rem *** Create joints ***
rem *** Struts to body ***
CreateWeldJoint(3,2,3,GetSpriteX(2),GetSpriteY(2),1)
CreateWeldJoint(4,3,5,GetSpriteX(5),GetSpriteY(5),1)

rem *** Struts to wheels ***
CreateRevoluteJoint(1,1,2,GetSpriteXByOffset(1),
    ↳GetSpriteYByOffset(1),0)
CreateRevoluteJoint(2,4,5,GetSpriteXByOffset(4),
    ↳GetSpriteYByOffset(4),0)

rem *** Set gravity ***
SetPhysicsGravity(0,2)

rem *** Run simulation ***
do
    Sync()
loop

```

Activity 20.41

Create a new project called *JointRevolute02*. Copy to the *media* folder the files *AGKDownloads/Chapter20/Wheel.png*, *Strut.png* and *Container.png*.

Implement and test the code given in FIG-20.97. What happens to the buggy?

Save your project.

SetJointMotorOn()

Our buggy from the last project isn't going to go anywhere if there is no motor to supply power to the wheels. And this is exactly what we can do using the `SetJointMotorOn()` statement. This statement can be used to supply a continuous turning force to revolute joints (and some other joints). The `SetJointMotorOn()` statement has the format shown in FIG-20.98.

FIG-20.98

SetJointMotorOn()

`SetJointMotorOn` (`id` , `fspeed` , `fmax`)

where

- | | |
|---------------|--|
| id | is an integer value giving the ID of the joint to which the motor is to be added. |
| fspeed | is an real value giving the desired speed of rotation. Use a negative number for clockwise rotation. |
| fmax | is the maximum force to be applied in order to achieve the desired speed. |

A motor may not achieve the required speed if frictional forces or other restraints are too large.

Activity 20.42

In project *JointRevolute02*, add the following lines immediately after all joints have been created:

```
rem *** Add motor to left wheel ***
SetJointMotorOn(1,-2,200)
```

Test the new code. What effect does this have on the buggy?

Modify your code so that after 15 seconds the motor reverses direction.

Save your project.

SetJointMotorOff()

A motor can be switched off using the `SetJointMotorOff()` statement which has the format shown in FIG-20.99.

FIG-20.99

SetJointMotorOff()

`SetJointMotorOff` (`id`)

where

- | | |
|-----------|---|
| id | is an integer value giving the ID of the joint which has been assigned the motor. |
|-----------|---|

Activity 20.43

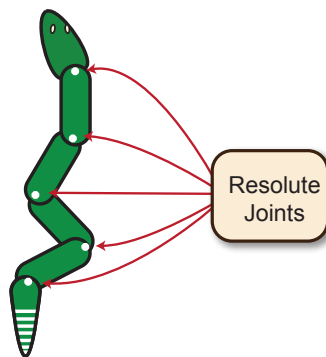
Modify *JointRevolute02* so that the motor is switched off after 25 seconds.

Test and save your project.

The final example of the revolute joint creates a chain of links to produce an effect similar to a segmented toy snake (see FIG-20.100).

FIG-20.100

Multiple Resolute Joints and Motors



Each of the body sprites has two revolute joints; one to the sprite ahead of it and one to the sprite behind it. In addition, the head and tail sprites (which each contain only a single joint) have motors attached to create some movement in the snake.

FIG-20.101

Using Motors on Resolute Joints

The model is demonstrated in the program listed in FIG-20.101.

```
rem *** A Revolute Joint Snake ***

rem *** Load Image ***
LoadImage(1,"SnakeHead.png")
LoadImage(2,"SnakeLink.png")
LoadImage(3,"SnakeTail.png")

rem *** Create sprites ***
rem *** Head ***
CreateSprite(1,1)
SetSpriteSize(1,4,-1)
SetSpritePosition(1,48,10)
rem *** Body ***
CreateSprite(2,2)
SetSpriteSize(2,4,-1)
SetSpritePosition(2,48,15)
CloneSprite(3,2)
SetSpritePosition(3,48,22.5)
CloneSprite(4,2)
SetSpritePosition(4,48,30)
CloneSprite(5,2)
SetSpritePosition(5,48,37.5)
rem *** Tail
CreateSprite(6,3)
SetSpriteSize(6,4,6)
SetSpritePosition(6,48,45)

rem *** Set background colour ***
SetClearColor(120,120,120)
rem *** Switch on physics ***
SetSpritePhysicsOn(1,2) //Dynamic
SetSpritePhysicsOn(2,2) //Dynamic
SetSpritePhysicsOn(3,2) //Dynamic
SetSpritePhysicsOn(4,2) //Dynamic
SetSpritePhysicsOn(5,2) //Dynamic
SetSpritePhysicsOn(6,2) //Dynamic
```



FIG-20.101

Using Motors on Resolute Joints

```
rem *** Create joints between each segment of the snake ***
CreateRevoluteJoint(1,1,2,GetSpriteXByOffset(1),GetSpriteY(1)+
↳GetSpriteHeight(1),0)
CreateRevoluteJoint(2,2,3,GetSpriteXByOffset(2),GetSpriteY(2)+
↳GetSpriteHeight(2),0)
CreateRevoluteJoint(3,3,4,GetSpriteXByOffset(3),GetSpriteY(3)+
↳GetSpriteHeight(3),0)
CreateRevoluteJoint(4,4,5,GetSpriteXByOffset(4),GetSpriteY(4)+
↳GetSpriteHeight(4),0)
CreateRevoluteJoint(5,5,6,GetSpriteXByOffset(5),GetSpriteY(5)+
↳GetSpriteHeight(5),0)
rem *** Switch on motors in the head and tail segments ***
SetJointMotorOn(1,-4,400)
SetJointMotorOn(5,4,400)

rem *** Set gravity ***
SetPhysicsGravity(0,1)

rem *** Run simulation ***
time = GetSeconds()
do
    if GetSeconds() > time
        SetJointMotorOn(1,Random(0,8)-4,800)
        SetJointMotorOn(5,Random(0,8)-4,800)
        time = GetSeconds()
    endif
    Sync()
loop
```

Activity 20.44

Create a new project called *JointRevolute03*. Copy the files *AGKDownloads/Chapter20/SnakeHead.png*, *SnakeLink.png*, *SnakeTail.png* into the project's *media* folder. Implement the code given in FIG-20.101.

Test your project. What happens when a sprite collides with a non-adjacent sprite?

Save your project.

As you have seen from the Activity above, when a sprite rotates into its neighbour, they pass by each other without colliding. This is because we selected not to allow collisions when creating the revolute joints (as specified in the final, zero, parameter in the function call). But collisions between sprites that are not connected directly are dealt with in the usual way, meaning that those sprites cannot slip past each other.

SetJointLimitOn()

The movement of the snake's head and tail appear unrealistic because they are able to perform a full rotation. In a real toy, the degree to which each segment can rotate about its neighbour would be limited. We can emulate this restriction using the `SetJointLimitOn()` statement which allows us to specify a limit to the degree of rotation allowed. The `SetJointLimitOn()` statement has the format shown in FIG-20.102.

FIG-20.102

SetJointLimitOn()

A syntax diagram for the SetJointLimitOn() function. It shows the function name in a rounded rectangle, followed by an opening parenthesis, then the parameter 'id' in a box, a comma, the parameter 'flower' in a box, a comma, the parameter 'fupper' in a box, and a closing parenthesis.

where

id is an integer value giving the ID of the joint whose movement is to be restricted

flower is a real value giving the lowest displacement allowed.

fupper is a real number giving the highest displacement allowed.

Since the motor is creating a rotational movement, *flower* gives the maximum angle to which the joint can rotate in a counterclockwise direction (given as a negative value) and *fupper* gives the maximum angle allowed in a clockwise direction.

Both angles are given in radians.

Activity 20.45

Modify *JointRevolute03* by limiting the rotation of the head to ± 2 radians and the tail to ± 2.5 radians.

Add the required lines immediately after the motors are activated.

Test and save your project.

SetJointLimitOff()

A joint limit can be deactivated using the `SetJointLimitOff()` statement (see FIG-20.103).

FIG-20.103

SetJointLimitOff()

A syntax diagram for the SetJointLimitOff() function. It shows the function name in a rounded rectangle, followed by an opening parenthesis, the parameter 'id' in a box, and a closing parenthesis.

where

id is an integer value giving the ID of the joint whose motor's limit is to be removed.

CreatePrismaticJoint()

A prismatic joint is one which allows two sprites to separate and come together along a fixed axis. Typical real-world examples of this type of movement are a train moving along a straight track, the plunger of a syringe being pushed in or pulled out, or an elevator moving up and down within the elevator shaft.

To create a prismatic joint between two sprites in AGK, we make use of the `CreatePrismaticJoint()` statement (see FIG-20.104).

FIG-20.104 CreatePrismaticJoint()**Format 1**

A syntax diagram for the CreatePrismaticJoint() function in Format 1. It shows the function name in a rounded rectangle, followed by an opening parenthesis, then 'id' in a box, a comma, 'sprld1' in a box, a comma, 'sprld2' in a box, a comma, 'x1' in a box, a comma, 'y1' in a box, a comma, 'x2' in a box, a comma, 'y2' in a box, a comma, 'icold' in a box, and a closing parenthesis.

Format 2

integer
 A syntax diagram for the CreatePrismaticJoint() function in Format 2. It shows the function name in a rounded rectangle, followed by an opening parenthesis, then 'sprld1' in a box, a comma, 'sprld2' in a box, a comma, 'x1' in a box, a comma, 'y1' in a box, a comma, 'x2' in a box, a comma, 'y2' in a box, a comma, 'icold' in a box, and a closing parenthesis.

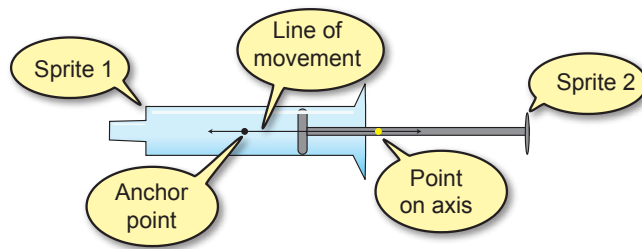
where

id	is an integer value giving the ID to be assigned to the joint.
sprId1	is an integer value giving the ID of the first sprite involved in the joint.
sprId2	is an integer value giving the ID of the second sprite involved in the joint.
x1,y1	are real values giving the coordinates of the anchor point for the first sprite.
x2,y2	are real values giving the coordinates of a point along the axis along which the sprites can move. These are measured relative to the anchor point.
icold	is an integer value specifying whether the sprites may collide.

FIG-20.105

A typical setup for this joint is shown in FIG-20.105.

A Typical Prismatic Joint



In this example, the centre of sprite 1 is given as the anchor point and the second point is offset in the *x* direction only, making the line of movement allowed horizontal. This setup could be achieved with the line:

```
CreatePrismaticJoint(1,1,2,GetSpriteXByOffset(1),  
↳GetSpriteYByOffset(1),1,0,0)
```

A force will be required to move the sprites. This force could be gravity, the result of a collision, or by applying a force directly to one of the sprites. However, another option is to add a motor to one of the sprites using the `SetJointMotorOn()` statement. Of course, in this case, the motor will create a linear movement rather than a circular one.

The program in FIG-20.106 demonstrates the use of this joint in the setup shown above. A motor is applied to sprite 2 (the plunger) to create movement.

FIG-20.106

Using a Prismatic Joint

```
rem *** A Prismatic Joint ***  
  
rem *** Load Image ***  
LoadImage(1,"Tube.png")  
LoadImage(2,"Plunger.png")  
  
rem *** Create sprites ***  
rem *** Tube ***  
CreateSprite(1,1)  
SetSpriteSize(1,14,-1)  
SetSpritePosition(1,43,40)
```



FIG-20.106

(continued)

Using a Prismatic Joint

Using a negative value for the second parameter of the `SetJointMotorOn()` statement will make the plunger move in the opposite direction.

```
rem *** Plunger ***
CreateSprite(2,2)
SetSpriteSize(2,14,-1)
SetSpritePosition(2,45.5,41)

rem *** Set background colour ***
SetClearColor(120,120,120)

rem *** Switch on physics ***
SetSpritePhysicsOn(1,1) //Static
SetSpritePhysicsOn(2,2) //Dynamic

rem *** Create a joint between tube and plunger ***
CreatePrismaticJoint(1,1,2,GetSpriteXByOffset(1),
    ↳GetSpriteYByOffset(1),1,0,0)

rem *** Switch on motor ***
SetJointMotorOn(1,0.2,400)

rem *** Set gravity ***
SetPhysicsGravity(0,0)

rem *** Run simulation ***
do
    Sync()
loop
```

Activity 20.46

Create a new project called *JointPrismatic*. Copy the files *AGKDownloads/Chapter20/Tube.png* and *Plunger.png* into the project's *media* folder.

Implement the code given in FIG-20.106.

Test your program. What happens when the plunger moves?

Save your project.

To stop the plunger moving all the way out of the tube, we can make use of the `SetJointLimitOn()` statement. This time the parameters of the statement are used to determine the distance that the sprite can move to either side of its starting position. Unfortunately, there is no obvious relationship between the parameter's value and the actual distance allowed, so you just have to try various values until you get one that suits the restraints you wish to apply.

Activity 20.47

Modify *JointPrismatic* by adding the lines

```
rem *** Limit movement ***
SetJointLimitOn(1,0,0.4)
```

Test your program. What happens when the plunger moves this time?

Save your project.

CreateLineJoint()

The line joint (also known as a **wheel** joint) is similar to a prismatic joint, with one sprite moving along a specified axis, but a line joint also allows that same sprite to rotate. FIG-20.107 shows an example of the type of movement allowed by this joint.

FIG-20.107

How a Line Joint Operates

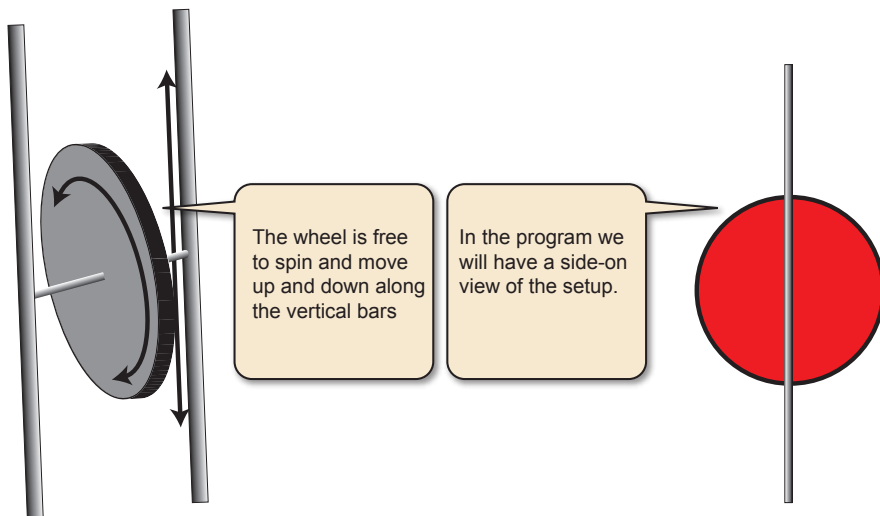


FIG-20.108

In AGK, a line joint is produced using the `CreateLineJoint()` statement (see FIG-20.108).

`CreateLineJoint()`

Format 1

`CreateLineJoint` (`id` , `sprld1` , `sprld2` , `x1` , `y1` , `x2` , `y2` , `icold`)

Format 2

integer `CreateLineJoint` (`sprld1` , `sprld2` , `x1` , `y1` , `x2` , `y2` , `icold`)

where

- | | |
|---------------|--|
| id | is an integer value giving the ID to be assigned to the joint. |
| sprld1 | is an integer value giving the ID of the first sprite involved in the joint. |
| sprld2 | is an integer value giving the ID of the second sprite involved in the joint. |
| x1,y1 | are real values giving the coordinates of the anchor point for the first sprite. |
| x2,y2 | are real values giving the coordinates of a point on the axis along which the sprites can move. These are measured relative to the anchor point. |
| icold | is an integer value specifying whether the sprites may collide. |

A program demonstrating the setup shown above is given in FIG-20.109.

FIG-20.109

Using a Line Joint

```
rem *** A Line Joint ***

rem *** Load Images ***
LoadImage(1,"Wheel2.png")
LoadImage(2,"Rail.png")

rem *** Create sprites ***
rem *** Wheel ***
CreateSprite(1,1)
SetSpriteSize(1,10,-1)
SetSpritePosition(1,45,40)
rem *** Rail ***
CreateSprite(2,2)
SetSpriteSize(2,0.5,40)
SetSpritePosition(2,50,20)

rem *** Switch on physics ***
SetSpritePhysicsOn(1,2) //Dynamic
SetSpritePhysicsOn(2,1) //Static

rem *** Make wheel's bounding box round ***
SetSpriteShape(1,1)

rem *** Create line joint ***
CreateLineJoint(1,2,1,GetSpriteXByOffset(1),
↳GetSpriteYByOffset(1),0,0.5,0)

rem *** Set gravity ***
SetPhysicsGravity(0,12)

rem *** Run simulation ***
do
    Sync()
loop
```

Activity 20.48

Create a new project called *JointLine*. Copy the files *AGKDownloads/Chapter20/Wheel2.png* and *Rail.png* into the project's *media* folder.

Implement the code given in FIG-20.109.

Test your program and observe the results.

Save your project.

You can add a motor to a line joint. This will cause the moveable sprite to rotate but does not affect its linear movement.

Activity 20.49

Modify *JointLine* by adding a `SetJointMotorOn()` statement immediately after `CreateLineJoint()` statement. The speed and power settings should be 10 and 200 respectively.

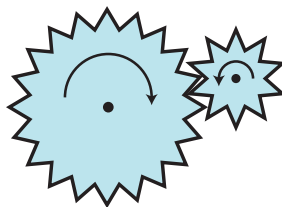
Test your program and observe the results. Save your project.

CreateGearJoint()

A gear joint creates a link between two existing revolute joints ensuring that both revolve in synchronisation with each other. The obvious example of a gear joint in the real world is a pair of gear wheels (see FIG-20.110).

FIG-20.110

How a Gear Joint Operates



A gear joint can be created using the `CreateGearJoint()` statement (see FIG-20.111).

FIG-20.111

`CreateGearJoint()`

Format 1

`CreateGearJoint` (`id` , `jntId1` , `jntId2` , `fratio`)

Format 2

integer `CreateGearJoint` (`jntId1` , `jntId2` , `fratio`)

where

- id** is an integer value giving the ID to be assigned to the gear joint.
- jntId1** is an integer value giving the ID of the first existing revolute joint to become part of the gear joint.
- jntId2** is an integer value giving the ID of the second existing revolute joint to become part of the gear joint.
- fratio** is a real number giving the ratio of the speeds of the two joints. For example, a value of 2 would mean that the revolving sprite attached to the first joint would spin twice as fast as the sprite attached to the second joint.

The program in FIG-20.112 uses a cog wheel sprite and a round sprite to create a revolute joint. The round sprite (positioned at the centre of the cog wheel) is then made invisible. A second, but smaller, cog wheel and joint is created alongside the first. These two joints are then linked to form a gear joint.

FIG-20.112

Using a Gear Joint

```
rem *** A Gear Joint ***

rem *** Load Images ***
LoadImage(1,"Cog1.png")
LoadImage(2,"Cog2.png")
LoadImage(3,"Shape02.png")

rem *** Create sprites ***
rem *** Large cog ***
CreateSprite(1,1)
SetSpriteSize(1,20,-1)
SetSpritePosition(1,45,10)
```



FIG-20.112

(continued)

Using a Gear Joint

```

rem *** Small cog ***
CreateSprite(2,2)
SetSpriteSize(2,13,-1)
SetSpritePosition(2,64,16.5)
rem *** Cog centres ***
CreateSprite(3,3)
SetSpriteSize(3,2,-1)
SetSpritePositionByOffset(3,GetSpriteXByOffset(1),
↳GetSpriteYByOffset(1))
CloneSprite(4,3)
SetSpritePositionByOffset(4,GetSpriteXByOffset(2),
↳GetSpriteYByOffset(2))

rem *** Switch on physics ***
SetSpritePhysicsOn(1,2) //Dynamic
SetSpritePhysicsOn(2,2) //Dynamic
SetSpritePhysicsOn(3,1) //Static
SetSpritePhysicsOn(4,1) //Static
rem *** Use round bounding boxes ***
SetSpriteShape(1,1)
SetSpriteShape(3,1)
SetSpriteShape(2,1)
SetSpriteShape(4,1)

rem *** Make sentral sprites invisible ***
SetSpriteVisible(3,0)
SetSpriteVisible(4,0)

rem *** Create revolute joints ***
CreateRevoluteJoint(1,3,1,GetSpriteXByOffset(1),
GetSpriteYByOffset(1),0)
CreateRevoluteJoint(2,4,2,GetSpriteXByOffset(2),
GetSpriteYByOffset(2),0)

rem *** Switch on motor for first cog ***
SetJointMotorOn(1,2,200)

rem *** Create gear joint ***
CreateGearJoint(3,1,2,4)

rem *** Run simulation ***
do
    Sync()
loop

```

Activity 20.50

Create a new project called *JointGear*. Copy the files *AGKDownloads/Chapter20/Cog1.png*, *Cog2.png* and *Shape02.png* into the project's *media* folder.

Implement the code given in FIG-20.112.

Test your program, observe the results and save your project.

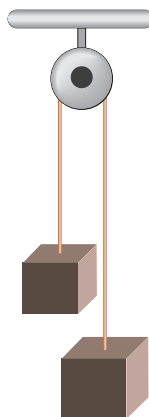
CreatePulleyJoint2() and FinishPulleyJoint()

Most people will have seen a pulley system at some time (see FIG-20.113).

FIG-20.113

A Typical Pulley System

Pulley System



In AGK BASIC, a pulley joint needs so many parameters that we need to use two separate functions to set one up.

FIG-20.114

The first of these is the `CreatePulleyJoint2()` (see FIG-20.114).

`CreatePulleyJoint2()`

`CreatePulleyJoint2` ((`sprId1` , `sprId2` , `fratio` , `icold`))

where

Just in case you were wondering, there is a `CreatePulleyJoint()` statement but it is only available when using AGK with C++.

- sprId1** is an integer value giving the ID of the first sprite.
- sprId2** is an integer value giving the ID of the second sprite.
- fratio** is a real number giving the movement ratio between the first and second sprites.
- icold** is an integer value (0 or 1) used to determine if the sprites collide or overlap when they come into contact with each other (0: overlap; 1: collide).

Once the `CreatePulleyJoint()` statement has been executed, this is followed by a call to `FinishPulleyJoint()` which has the format shown in FIG-20.115.

FIG-20.115

`FinishPulleyJoint()`

integer `FinishPulleyJoint` ((`id` , `gx1` , `gy1` , `gx2` , `gy2` , `x1` , `y1` , `x2` , `y2`))

where

- id** is an integer value giving the ID to be assigned to the completed pulley joint.
- gx1,gy1** are a pair of real values giving the ground point for the first sprite. This point determines the maximum point to which the first sprite can be lifted.
- gx2,gy2** are a pair of real values giving the ground point for the second sprite.

x1,y1 are a pair of real values giving the anchor point for the first sprite.

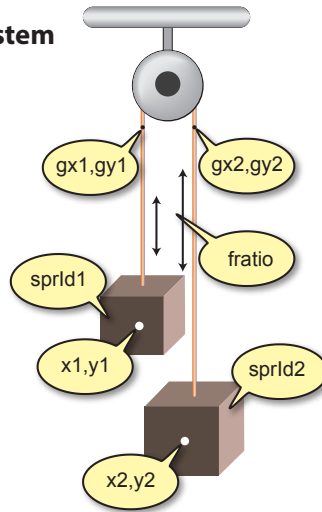
x2,y2 are a pair of real values giving the anchor point for the second sprite.

To help understand the purpose of each parameter for these two statements, typical points or values that would be used for each are shown in FIG-20.116.

FIG-20.116

Pulley Joint Parameters

Pulley System



The program in FIG-20.117 simulates this exact pulley setup. The only trick is to adjust the height of the rope sprites as the blocks on the pulley move.

FIG-20.117

Using a Pulley Joint

```
rem *** A Pulley Joint ***

rem *** Load Images ***
LoadImage(1,"Pulley.png")
LoadImage(2,"Box.png")
LoadImage(3,"Rope.png")

rem *** Create sprites ***
rem *** Pulley ***
CreateSprite(1,1)
SetSpriteSize(1,24,-1)
SetSpritePosition(1,38,10)
rem *** Box 1 ***
CreateSprite(2,2)
SetSpriteSize(2,10,-1)
SetSpritePosition(2,40,40)
SetSpriteDepth(2,12)
rem *** Box 2 ***
CloneSprite(3,2)
SetSpritePosition(3,50,50)
rem *** Rope 1 ***
CreateSprite(4,3)
SetSpriteSize(4,0.5,23)
SetSpritePosition(4,45,18)
SetSpriteDepth(4,11)
rem *** Rope 2 ***
CloneSprite(5,4)
SetSpriteSize(5,0.5,33)
SetSpritePosition(5,55,18)
```



FIG-20.117

(continued)

Using a Pulley Joint

```

rem *** Set background colour ***
ClearColor(120,120,120)
rem *** Switch on physics ***

SetSpritePhysicsOn(1,1) //Static
SetSpritePhysicsOn(2,2) //Dynamic
SetSpritePhysicsOn(3,2) //Dynamic

rem *** Create pulley joint ***
CreatePulleyJoint2(2,3,1.25,0)
id = FinishPulleyJoint(45,20,55,20,GetSpriteXByOffset(2),
↳GetSpriteYByOffset(2),GetSpriteXByOffset(3),
↳GetSpriteYByOffset(3))

rem *** Run simulation ***
do
    rem *** Adjust the length of both ropes ***
    heightrope1#=GetSpriteY(3)-GetSpriteY(5)+1
    heightrope2#= GetSpriteY(2)-GetSpriteY(4)+1
    SetSpriteSize(5,0.5,heightrope1#)
    SetSpriteSize(4,0.5,heightrope2#)
    Sync()
loop

```

Activity 20.51

Start a new project called *JointPulley*. Copy the necessary image files from *AGKDownloads/Chapter20/* to the project's *media* folder.

Set the app window's size to 678 by 1024.

Implement and test the code given in FIG-20.117.

Change the ratio value in `CreatePulleyJoint2()` from 1.25 to 0.75. How does this affect the result? Save your project.

GetJointReactionTorque()

We saw earlier when looking at the weld joint, that joints can be subjected to linear forces and that these are detected using the `GetJointReactionForceX()` and `GetJointReactionForceY()`. Joints may also experience turning forces and these can be detected using the `GetJointReactionTorque()` statement (see FIG-20.118).

FIG-20.118

GetJointReactionTorque()

float `GetJointReactionTorque` (`id`)

where

id is an integer value giving the ID of the joint to be tested.

GetJointExists()

To check if a joint of a specified ID currently exists, use the `GetJointExists()` statement (see FIG-20.119).

FIG-20.119

GetJointExists()

integer `GetJointExists` (`id`)

where

id is an integer value giving the ID to be tested.

The function returns 1 if the stated joint exists, otherwise zero is returned.

Summary

- Joints allow two or more physics-enabled sprites to be linked, setting up a specific relationship between their interaction.
- In a weld joint, two sprites are joined together in a fashion similar to magnetic attraction.
- Use `CreateWeldJoint()` to create a weld joint.
- Use `GetJointReactionForceX()` and `GetJointReactionForceY()` to find the x and y components of any force acting on a joint.
- Use `DeleteJoint()` to delete a joint.
- A distance joint keeps two points (one in each sprite) a fixed distance apart at all times.
- Use `CreateDistanceJoint()` to create a distance joint.
- A mouse joint is attracted to a specific point in space.
- Use `CreateMouseJoint()` to create a mouse joint.
- Use `SetMouseJointTarget()` to specify the point to which the mouse joint is attracted.
- A revolute joint allows one sprite to revolve about a point on a second sprite.
- Use `CreateRevoluteJoint()` to create a revolute joint.
- A motor can be attached to certain types of joints, giving that joint power to initiate movement.
- Use `SetJointMotorOn()` to power on a motor at a specified joint.
- Use `SetJointMotorOff()` to power off the motor at a specified joint.
- Use `SetJointLimitOn()` to limit the movement of a joint.
- Use `SetJointLimitOff()` to switch off any restriction on joint movement.
- A prismatic joint allows sprites to move along a fixed line relative to each other.
- Use `CreatePrismaticJoint()` to create a prismatic joint.
- A line joint allows sprites to move along a fixed line relative to each other and for the moving sprite to rotate.
- Use `CreateLineJoint()` to create a line joint.
- A gear joint links two existing revolute joints to ensure that the two rotating sprites move in relation to each other.
- Use `CreateGearJoint()` to create a gear joint.
- A pulley joint is one in which two sprites move in opposite directions to each

other in the same way as a pulley system.

- Use `CreatePulleyJoint2()` and `FinishPulleyJoint()` to create a pulley joint.
- Use `GetJointReactionTorque()` to find the turning force exerted on a joint.
- Use `GetJointExists()` to check that a joint exists.

Solutions

Activity 20.1

The ball speeds up as it falls - just as a real ball being pulled towards the ground by gravity.

When the ball hits the bottom of the app window it bounces slightly.

To change the sprite to a static object, the line

```
SetSpritePhysicsOn(1,2)
```

must be changed to

```
SetSpritePhysicsOn(1,1)
```

The ball does not move when the program is run.

To change the sprite to a kinematic object we must now change the line

```
SetSpritePhysicsOn(1,1)
```

to

```
SetSpritePhysicsOn(1,3)
```

Like the static object, the kinematic one does not move.

Activity 20.2

Modified code for *Physics01*:

```
rem *** Basic Physics ***

rem *** Load image ***
LoadImage(1,"Ball.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,6,-1)
SetSpritePosition(1,50,5)
rem *** Apply kinematic physics ***
SetSpritePhysicsOn(1,3)
rem *** Add a velocity ***
SetSpritePhysicsVelocity(1,15*cos(60),15*sin(60))
do
    Sync()
loop
```

Activity 20.3

No solution required.

Activity 20.4

Modified code for *Physics01*:

```
rem *** Basic Physics ***

rem *** Load image ***
LoadImage(1,"Ball.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,6,-1)
SetSpritePosition(1,50,5)
rem *** Apply dynamic physics ***
SetSpritePhysicsOn(1,2)
rem *** Set ball's bounce factor ***
SetSpritePhysicsRestitution(1,0.75)
rem *** Create text object ***
CreateText(1,"")
do
    rem *** Update text ***
    SetTextString(1,"Ball velocity Y : "+
        Str(GetSpritePhysicsVelocityY(1)))
    Sync()
loop
```

Activity 20.5

Modified code for *Physics02*:

```
rem *** Angular Velocity ***

rem *** Load image ***
LoadImage(1,"Tile.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,10,10)
rem *** Apply dynamic physics ***
SetSpritePhysicsOn(1,2)
rem *** Apply angular velocity ***
SetSpritePhysicsAngularVelocity(1,50)
do
    Sync()
loop
```

This time the tile falls while spinning at a faster rate than before.

Activity 20.6

Modified code for *Physics02*:

```
rem *** Angular Velocity ***

rem *** Load image ***
LoadImage(1,"Tile.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,40,10)
rem *** Apply kinematic physics ***
SetSpritePhysicsOn(1,2)
rem *** Apply angular velocity ***
SetSpritePhysicsAngularVelocity(1,50)
rem *** Apply angular damping ***
SetSpritePhysicsAngularDamping(1,0.7)
do
    Sync()
loop
```

The angular damping reduces the speed of the spin as the tile falls.

Activity 20.7

Modified code for *Physics02*:

```
rem *** Torque ***

rem *** Load image ***
LoadImage(1,"Tile.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,40,10)
rem *** Apply dynamic physics ***
SetSpritePhysicsOn(1,2)
do
    rem *** Apply torque ***
    SetSpritePhysicsTorque(1,200)
    Sync()
loop
```

With such a high torque, the sprite continues to spin even after hitting the ground.

Activity 20.8

Modified code for *Physics02*:

```
rem *** Torque ***

rem *** Create text ***
CreateText(1,"")
SetTextPosition(1,5,5)
SetTextSize(1,3)
rem *** Load image ***
LoadImage(1,"Tile.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,40,10)
```

```

rem *** Apply dynamic physics ***
SetSpritePhysicsOn(1,2)
do
    rem *** Apply torque ***
    SetSpritePhysicsTorque(1,200)
    rem *** Display angular velocity ***
    SetTextString(1,"Angular velocity: "+
        ⚡Str(GetSpritePhysicsAngularVelocity(1)))
    Sync()
loop

```

Activity 20.9

Modified code for *Physics02*:

```

rem *** Torque ***

rem *** Create text ***
CreateText(1,"")
SetTextPosition(1,5,5)
SetTextSize(1,3)
rem *** Load image ***
LoadImage(1,"Tile.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,15,-1)
SetSpritePosition(1,40,10)
rem *** Apply dynamic physics ***
SetSpritePhysicsOn(1,2)
rem *** Switch off rotation ***
SetSpritePhysicsCanRotate(1,0)
do
    rem *** Apply torque ***
    SetSpritePhysicsTorque(1,200)
    rem *** Display angular velocity ***
    SetTextString(1,"Angular velocity: "+
        ⚡Str(GetSpritePhysicsAngularVelocity(1)))
    Sync()
loop

```

The sprite falls without spinning.

Activity 20.10

When the ship reaches the top of the window it stops.

Modified code for *Physics03 (1)*:

```

rem *** Force to Right ***

rem *** Load image ***
LoadImage(1,"Rocketship.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,-1,20)
SetSpritePosition(1,45,80)
rem *** Turn on physics for sprite ***
SetSpritePhysicsOn(1,2)
do
    rem *** Apply horizontal force to the right ***
    SetSpritePhysicsForce(1,GetSpriteX(1)+
        ⚡GetSpriteWidth(1)/2.0, GetSpriteY(1)+
        ⚡GetSpriteHeight(1)/2.0,200,0)
    Sync()
loop

```

The new force causes the ship to move to the side and topple over. There is a turning force on the craft caused by friction with the ground.

Modified code for *Physics03 (2)*:

```

rem *** Force Upwards ***

rem *** Load image ***
LoadImage(1,"Rocketship.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,-1,20)
SetSpritePosition(1,45,80)
rem *** Turn on physics for sprite ***
SetSpritePhysicsOn(1,2)
rem *** Single application of force ***
SetSpritePhysicsForce(1,GetSpriteX(1)+
    ⚡GetSpriteWidth(1)/2.0, GetSpriteY(1)+
    ⚡GetSpriteHeight(1)/2.0,0,-20000)
do

```

```

Sync()
loop

```

The craft lifts and then falls back to the ground.

Activity 20.11

Modified code for *Physics03(1)*:

```

rem *** Force Upwards ***

rem *** Load image ***
LoadImage(1,"Rocketship.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,-1,20)
SetSpritePosition(1,45,80)
rem *** Turn on physics for sprite ***
SetSpritePhysicsOn(1,2)
rem *** Single application of force ***
SetSpritePhysicsForce(1,GetSpriteX(1)+
    ⚡GetSpriteWidth(1)/2.0, GetSpriteY(1)+
    ⚡GetSpriteHeight(1)/2.0,0,-2000)
do
    Sync()
loop

```

The craft will not lift off the ground.

Modified code for *Physics03(2)*:

```

rem *** Force Upwards ***

rem *** Load image ***
LoadImage(1,"Rocketship.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,-1,20)
SetSpritePosition(1,45,80)
rem *** Turn on physics for sprite ***
SetSpritePhysicsOn(1,2)
rem *** Single application of force ***
SetSpritePhysicsImpulse(1,GetSpriteX(1)+
    ⚡GetSpriteWidth(1)/2.0, GetSpriteY(1)+
    ⚡GetSpriteHeight(1)/2.0,0,-2000)
do
    Sync()
loop

```

This time the nose of the craft lifts to about 3/4 of the way up the window.

Activity 20.12

Modified code for *Physics03(1)*:

```

rem *** Force Upwards ***

rem *** Create text object ***
CreateText(1,"")
SetTextPosition(1,5,5)
SetTextSize(1,3)
rem *** Load image ***
LoadImage(1,"Rocketship.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,-1,20)
SetSpritePosition(1,45,80)
rem *** Turn on physics for sprite ***
SetSpritePhysicsOn(1,2)
rem *** Display mass ***
SetTextString(1,Str(GetSpritePhysicsMass(1)))
rem *** Single application of force ***
SetSpritePhysicsImpulse(1,GetSpriteX(1)+
    ⚡GetSpriteWidth(1)/2.0,GetSpriteY(1)+
    ⚡GetSpriteHeight(1)/2.0,0,-2000.0)
do
    Sync()
loop

```

The mass reading is 3.75.

Modified code for *Physics03(2)*:

```

rem *** Force Upwards ***

rem *** Create text object ***

```

```

CreateText(1,"")
SetTextPosition(1,5,5)
SetTextSize(1,3)
rem *** Load image ***
LoadImage(1,"Rocketship.png")
rem *** Create sprite ***

CreateSprite(1,1)
SetSpriteSize(1,-1,20)
SetSpritePosition(1,45,80)
rem *** Turn on physics for sprite ***
SetSpritePhysicsOn(1,2)
rem *** Increase mass 10 fold ***
SetSpritePhysicsMass(1,GetSpritePhysicsMass(1)*10)
rem *** Display mass ***
SetTextString(1,Str(GetSpritePhysicsMass(1)))
rem *** Single application of force ***
SetSpritePhysicsImpulse(1,GetSpriteX(1)+
    ↵GetSpriteWidth(1)/2.0,GetSpriteY(1)+
    ↵GetSpriteHeight(1)/2.0,0,-2000.0)
do
    Sync()
loop

```

The nose of the craft lifts to about half way up the screen.

Activity 20.13

To create a horizontal force, swap the positions of the last two parameters of `SetSpritePhysicsImpulse()`, changing the line from

```

SetSpritePhysicsImpulse(1,GetSpriteX(1)+
GetSpriteWidth(1)/2.0,GetSpriteY(1)+
GetSpriteHeight(1)/2.0,0,-2000.0)

```

to

```

SetSpritePhysicsImpulse(1,GetSpriteX(1)+
GetSpriteWidth(1)/2.0,GetSpriteY(1)+
GetSpriteHeight(1)/2.0,-2000.0,0)

```

The craft topples as it moves off to the left.

Modified code for *Physics03* with a friction value of 0.25:

```

rem *** Force to the Left ***

rem *** Create text object ***
CreateText(1,"")
SetTextPosition(1,5,5)
SetTextSize(1,3)
rem *** Load image ***
LoadImage(1,"Rocketship.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,-1,20)
SetSpritePosition(1,45,80)
rem *** Turn on physics for sprite ***
SetSpritePhysicsOn(1,2)
rem *** Increase mass 10 fold ***
SetSpritePhysicsMass(1,GetSpritePhysicsMass(1)*10)
rem *** Display mass ***
SetTextString(1,Str(GetSpritePhysicsMass(1)))
rem *** Set friction ***
SetSpritePhysicsFriction(1,0.25)
rem *** Single application of force ***
SetSpritePhysicsImpulse(1,GetSpriteX(1)+
    ↵GetSpriteWidth(1)/2.0,GetSpriteY(1)+
    ↵GetSpriteHeight(1)/2.0,-2000.0,0)
do
    Sync()
loop

```

The craft now slides to the side without toppling. A friction value as low as 0.28 causes the craft to fall over.

To set the friction to zero, change the `SetSpritePhysicsFriction()` statement to read:

```
SetSpritePhysicsFriction(1,0)
```

With friction set to zero the craft slams into the left edge of the app window and then starts to move to the right. With no friction it continues moving at the same speed until it hits the right edge and then slowly starts to move left again.

Activity 20.14

Modified code for *Physics03*:

```

rem *** Zero friction / 0.2 Damping ***

rem *** Create text object ***
CreateText(1,"")
SetTextPosition(1,5,5)
SetTextSize(1,3)
rem *** Load image ***
LoadImage(1,"Rocketship.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,-1,20)
SetSpritePosition(1,45,80)
rem *** Turn on physics for sprite ***
SetSpritePhysicsOn(1,2)
rem *** Increase mass 10 fold ***
SetSpritePhysicsMass(1,GetSpritePhysicsMass(1)*10)
rem *** Display mass ***
SetTextString(1,Str(GetSpritePhysicsMass(1)))
rem *** Set friction ***
SetSpritePhysicsFriction(1,0)
rem *** Set damping ***
SetSpritePhysicsDamping(1,0.2)
rem *** Single application of force ***
SetSpritePhysicsImpulse(1,GetSpriteX(1)+
    ↵GetSpriteWidth(1)/2.0,GetSpriteY(1)+
    ↵GetSpriteHeight(1)/2.0,-2000.0,0)
do
    Sync()
loop

```

With damping at 0.2 the craft slows to a stop after hitting the left side.

Activity 20.15

No solution required.

Activity 20.16

Modified code for *Physics04*:

```

rem *** Physics collisions ***

rem *** Load images ***
LoadImage(1,"Bat.png")
LoadImage(2,"Ball.png")
rem *** Create text resource ***
CreateText(1,"")
rem *** Set up bat ***
CreateSprite(1,1)
SetSpriteSize(1,12,-1)
SetSpritePosition(1,44,48)
rem *** Make bat kinematic ***
SetSpritePhysicsOn(1,3)
rem *** Spin bat ***
SetSpritePhysicsAngularVelocity(1,100)
rem *** Set up ball ***
CreateSprite(2,2)
SetSpriteSize(2,6,-1)
SetSpritePosition(2,47,0)
rem *** Make ball dynamic ***
SetSpritePhysicsOn(2,2)
do
    if GetPhysicsCollision(1,2)
        SetTextString(1,"Hit")
    endif
    Sync()
loop

```

Activity 20.17

The `do...loop` within *Physics04* must be changed to:

```

do
    if GetPhysicsCollision(1,2)
        SetTextString(1,"Collision at (" +
            ↵Str(GetPhysicsCollisionX()) + ", " +
            ↵Str(GetPhysicsCollisionY()) + ")")
    endif
    Sync()
loop

```

Activity 20.18

The `do..loop` within *Physics04* must be changed to:

```

do
    if GetPhysicsCollision(1,2)
        SetTextString(1,"Collision at (" +
            ⤵Str(GetPhysicsCollisionWorldX())+"", " +
            ⤵Str(GetPhysicsCollisionWorldY())+"")
    endif
    Sync()
loop

```

Activity 20.19

In *Physics04*, add the line

```
SetSpritePhysicsDebugOn()
```

immediately before the

```
do .. loop
```

Activity 20.20

No solution required.

Activity 20.21

To have the screen represent a 100m x 100m area, change the line

```
SetPhysicsScale(0.5)
```

to

```
SetPhysicsScale(1.0)
```

To change to lunar gravity (1.63m/sec²), we need to add the lines

```
rem *** Set lunar gravity ***
SetPhysicsGravity(0,1.63)
```

immediately after the physics is switched on.

To make the screen represent a 25m x 25m area, we need to change the `SetPhysicsScale()` statement to read

```
SetPhysicsScale(0.25)
```

We also need to take this rescaling into account when setting the gravity, so the `SetPhysicsGravity()` statement must change to

```
SetPhysicsGravity(0,1.63/0.25)
```

Activity 20.22

To change the gravitational pull and remove the appropriate physics walls, use the lines

```
SetPhysicsGravity(1.63/0.25,-1.63/0.25)
rem *** Remove top and right walls ***
SetPhysicsWallTop(0)
SetPhysicsWallRight(0)

```

Activity 20.23

Change the call to `CreatePhysicsForce()` to read

```
forceid = CreatePhysicsForce(50,50,-4,20,15,1)
```

This will cause the sphere to be repelled by the moon.

Activity 20.24

No solution required.

Activity 20.25

If several sprites hit at the same time, only the tile in the first collision is reduced in size.

Activity 20.26

The new code ensures that all collisions are handled and so every contact results in a reduction in size of the tiles involved.

Activity 20.27

No solution required.

Activity 20.28

No solution required.

Activity 20.29

No solution required.

Activity 20.30

To have the oranges collide with the green apples after 10 seconds, change the end of the program to:

```

rem *** Record start time ***
time = GetSeconds()
do
    rem *** If 10 seconds passed ***
    if GetSeconds() - time = 10
        rem *** Set all oranges to collide with
        ⤵green apples ***
        for c = 11 to 20
            SetSpriteCollideBit(c,4,1)
        next c
    endif
    Sync()
loop#

```

The code sets bit 4 (the one assigned to green apples) in the hit categories for every orange.

Activity 20.31

No solution required.

Activity 20.32

Although the two sprites stay firmly attached, you may see a little movement in their relative positions when various forces act on the joint.

Activity 20.33

No solution required.

Activity 20.34

With gravity at 120, you should see a little more separation of the two sprites as the forces involved become larger.

With a setting of 1200, gravity becomes so strong that the two sprites may separate as they hit the triangle, but they will then snap together again.

Activity 20.35

The initial gap between the two sprites is maintained by the weld joint.

Activity 20.36

No solution required.

Activity 20.37

To set the centre of each sprite as an anchor point, change the line

```
CreateDistanceJoint(1,1,2,GetSpriteX(1),  
↳GetSpriteY(1),GetSpriteX(2),GetSpriteY(2),1)
```

to

```
CreateDistanceJoint(1,1,2,GetSpriteXByOffset(1),  
↳GetSpriteYByOffset(1),GetSpriteXByOffset(2),  
↳GetSpriteYByOffset(2),1)
```

Activity 20.38

The anchor point at the centre of the sprite is drawn towards the target point at the bottom right of the screen.

Activity 20.39

The sprite will roll towards the selected point. Gravity stops the ball lifting. However, if you were to switch off gravity, the sprite would move towards any point on the screen.

Activity 20.40

The bar does not move because it is perfectly balanced on the pivot.

When the position of the bar is changed, its right side is heavier than the left and so the bar rotates about the pivot point until the two sprites collide.

To stop the sprites bar and pivot sprites colliding, change the last parameter of the `CreateRevoluteJoint()` to zero.

This causes the bar to continue rotating about the pivot point; first in one direction and then the other.

Activity 20.41

The buggy falls a short distance to the bottom of the window.

Activity 20.42

Adding the motor gives power to the revolute joint and this powers the buggy moving it towards the right edge of the window.

To get the motor to reverse direction after 15 seconds, change the *run simulation* part of the code to read

```
rem *** Run simulation ***  
rem *** Record current time ***  
time = GetSeconds()  
rem *** Motor not reversed ***  
reversed = 0  
do  
    rem *** After 15 seconds, reverse direction ***  
    if GetSeconds()-time = 15 and reversed = 0  
        SetJointMotorOn(1,2,200)  
        reversed = 1  
    endif  
    Sync()  
loop
```

Activity 20.43

To make the motor switch off after 25 seconds, change the *run simulation* part of the code to read

```
rem *** Run simulation ***  
rem *** Record current time ***  
time = GetSeconds()  
rem *** Motor not reversed ***  
reversed = 0
```

```
do  
    rem *** After 15 seconds, reverse direction ***  
    if GetSeconds()-time = 15 and reversed = 0  
        SetJointMotorOn(1,2,200)  
        reversed = 1  
    endif  
    rem *** Switch motor off after 25 seconds ***  
    if GetSeconds()-time = 25  
        SetJointMotorOff(1)  
    endif  
    Sync()  
loop
```

The buggy will continue to move even after the motor has stopped because of low friction and damping.

Activity 20.44

Non-adjacent sprites contacts cause a collision event and the sprites involved cannot flow past each other.

Activity 20.45

To limit the movement of the head and tail sections, add the lines

```
rem *** Limit the movement of ***  
rem *** the head and tail sections ***  
SetJointLimitOn(1,-2,2) //Head  
SetJointLimitOn(5,-2.5,2.5) //Tail
```

before the *run simulation* section of the code.

Activity 20.46

The plunger moves to the right, hitting the edge of the screen.

Activity 20.47

The plunger now stops when it reaches the end of the syringe.

Activity 20.48

No solution required.

Activity 20.49

By adding the lines

```
rem *** Activate motor ***  
SetJointMotorOn(1,10,200)
```

you will see the wheel sprite rotate as it moves along the other sprite.

Activity 20.50

No solution required.

Activity 20.51

Initially, the left box falls and the right box rises. When the ratio value is changed from 1.25 to 0.75, it is the right box that falls and the left box that rises.

Accessing a Network

In this Chapter:

- ☐ Hardware Requirements for Networking
- ☐ Host and Client Machines
- ☐ Joining a Network
- ☐ Client Names and IDs
- ☐ Handling Devices on the Network
- ☐ Sending and Receiving Messages
- ☐ Local Client Data
- ☐ Transmission Times and Delays
- ☐ Broadcasting
- ☐ Creating Multiplayer Games
- ☐ Using HTTP

Multiplayer Games

Introduction

Perhaps most games are played in single-player mode; you against the computer. But a more enjoyable option is to play against another human player. After all, where's the fun in beating a piece of software?

Linking computers together gives us a lot more scope for a game; we can hide information from other players, the players can be half a world apart and many players can take part in a single game (assuming the game lends itself to multiple players).

Hardware Requirements

To get computers to communicate with each other, they need to be connected in some way. If only two machines are involved and they are located in the same building, then a simple Wi-Fi signal from your router will be sufficient. Over a longer distance, two machines could communicate using a phone line and routers.

A local area network (LAN) is one where the computers are in close physical proximity to each other; perhaps in the same room, building or campus.

A wide area network (WAN) is one where the linked computers are spread over greater distances, possibly even in different countries.

In setting up a network, there are choices to be made as to how the machines are to communicate with each other.

In a peer-to-peer network, each machine has equal status. Information from one machine is sent along a common connection and collected by the machine for which it was destined. Every machine in the network has its own unique address and the addresses of both the source computer and destination computer are sent as part of the information transmitted.

In a client/server setup, one machine acts as a server and the others as clients. The server has links to every client. While the server can communicate directly with any client, communication between clients must be routed through the server. The client machines send requests for data to the server and the server sends back the necessary information.

The Host and its Clients

In AGK, setting up a multiplayer game requires a network of communicating devices. This network is assigned a unique ID to ensure that any devices transmitting on a separate network do not cause interference. The individual devices within your network are also assigned a unique ID of their own. This ensures, amongst other things, that data transmitted over the network can be received by the correct device.

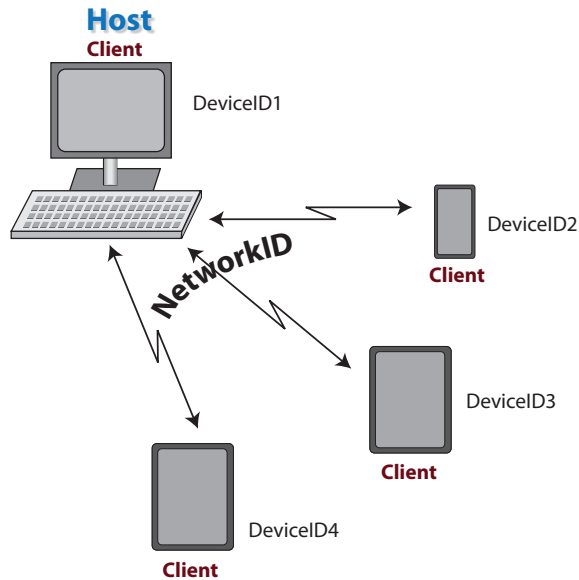
One of the machines on the network has to act as the **host** (also known as the **server**). The host machine is responsible for the overall control of the network. If the host machine shuts down, then the transmission of data between the linked devices will be terminated. Although a multiplayer game session only ever has one host device, there may be many others machines (clients) linked to the host. All machines within

an AGK network - including the host - are known as **clients**.

The basic setup of a game network is shown in FIG-21.1.

FIG-21.1

A Typical Wi-Fi
Network



Multiplayer Statements

HostNetwork()

The first requirement of a multiplayer game is to set up the host device. This is done using the `HostNetwork()` statement (see FIG-21.2).

FIG-21.2

HostNetwork()

integer `HostNetwork` ((`snetName` , `scient` , `iport`)

where

snetName	is a string giving the name to be used for the network hosting the game. Any name can be selected.
scient	is a string giving the name to be assigned to this client. Any unique name can be used, but no two clients may be assigned the same name.
iport	is an integer value (1026 to 65535) giving the port to be used to access the game session. If the host machine is not capable of producing a Wi-Fi broadcast, then it will communicate via this port. For example, a PC will normally communicate through the port to your Wi-Fi router.

The function returns the integer ID assigned to the network.

A typical statement to set up the host machine might be:

```
netid = HostNetwork("MyNetwork", "Hostmachine", 1026)
```

The short program in FIG-21.3 sets the computer up as the host machine for a network and displays the ID assigned to that network.

FIG-21.3

Setting up a Host

```
rem *** Host network ***
netid = HostNetwork("MyNetwork","Hostmachine",1026)

rem *** Display network's ID ***
Print("Network's ID is " + Str(netid))
Sync()

do
loop
```

Activity 21.1

Create a new project, *Multiplayer01*, and implement the code given in FIG-21.3.

What ID is assigned to the network?

Save the project.

IsNetworkActive()

To check that a network of a specified ID is currently active, we can use the `IsNetworkActive()` statement (see FIG-21.4).

FIG-21.4

`IsNetworkActive()`

integer `IsNetworkActive` ((`id`))

where

id is an integer value giving the ID assigned to the network by the `HostNetwork()` statement.

The statement will return 1 when run on the host machine if a network setup was successfully achieved using the `HostNetwork()` statement. The statement will return 1 when run on a client machine that has joined the network.

JoinNetwork()

Since a multiplayer app can have only one host, other devices must join using the `JoinNetwork()` statement. This statement has two different formats depending on whether you are joining your local LAN (format 1) or connecting over the internet (format 2) (see FIG-21.5).

FIG-21.5

`JoinNetwork()`

Format 1

integer `JoinNetwork` ((`snetName` , `sclient`))

Format 2

integer `JoinNetwork` ((`sIP` , `iport` , `sclient`))

where

snetName is a string giving the name to be used by the network

hosting the game. This must be the same value as used in the `HostNetwork()` statement.

sclient	is a string giving the name to be assigned to this client. Any unique name can be used but no two clients may be assigned the same name.
sIP	is a string giving the IP address of the host machine.
iport	is an integer value (1026 to 65535) giving the port to be accessed on the host machine.

When creating your app, it's quite possible for the host machine to run different code than that executed on the client devices, but it is no problem to run exactly the same code on all machines as long as there is some mechanism for making one machine the host and the others clients.

The program in FIG-21.6 demonstrates the basic ideas involved in setting up a connection between two machines, both running the same code. The program uses the following logic:

```
Display Host and Join buttons
DO
  IF the Host button is pressed THEN
    Host the network
  ENDIF
  IF the Join button is pressed THEN
    Join as client
  ENDIF
  IF connected to the network THEN
    Display active status
  ENDIF
LOOP
```

FIG-21.6

Creating a Network

```
rem *** Testing Multiplayer Statements ***

rem *** Create Host/Join buttons ***
AddVirtualButton(1,10,20,10)
SetVirtualButtonText(1,"Host")
AddVirtualButton(2,30,20,10)
SetVirtualButtonText(2,"Join")
rem *** Not yet joined the network ***
joined = 0
do
  rem *** If Host button pressed and not already joined ***
  if GetVirtualButtonPressed(1)= 1 and joined = 0
    rem *** Host the app ***
    netid = HostNetwork("MyNetwork","Hostmachine",1026)
    rem *** Record as joined ***
    joined = 1
  endif
  rem *** If Join button pressed and not already joined ***
  if GetVirtualButtonPressed(2)=1 and joined = 0
    rem *** Join as client ***
    netid = JoinNetwork("MyNetwork","Client1")
    rem *** Record as joined ***
    joined = 1
  endif
endif
```



FIG-21.6

(continued)

Creating a Network

```
rem *** If joined, check network is active ***
if joined = 1
    if IsNetworkActive(netid)
        Print("Connected")
    else
        Print("Not connected")
    endif
endif
rem *** Display the network's ID ***
Print("Network ID : "+Str(netid))
Sync()
loop
```

Activity 21.2

Start a new project called *MultiPlayer02* and implement the code shown in FIG-21.6.

We now need to run the program on two devices.

Run the AGK Player on your tablet or phone. Hit the **Compile/Run/Broadcast** button in the AGK editor.

The app should now appear on your computer and on your second device.

Click on the **Host** button on the computer's version of the app and on the **Join** button on your second device.

Both devices should now display a Connected message.

Close down the app on the main computer. What happens to the second device?

Save your project.

The code used in Activity 21.2 only allows two machines to be linked. The reason for this may not be immediately obvious, but the problem is the line

```
netid = JoinNetwork("MyNetwork", "Client1")
```

This assigns the name *Client1* to any machine joining the network as a client. Since no two machines may use the same name, an attempt by a third machine to join the network will fail.

A better version of the code would be

```
netid = JoinNetwork("MyNetwork", Str(GetSeconds()))
```

which will assign the time since startup as the client's name. This way, each machine can be assigned, its own unique identity.

Activity 21.3

Modify *Multiplayer02* as suggested above. Save the project.

GetNetworkNumClients()

The number of machines connected to a network can be discovered using the `GetNetworkNumClients()` (see FIG-21.7).

FIG-21.7

`GetNetworkNumClients()` integer `GetNetworkNumClients` ((id))

where

id is an integer value giving the ID assigned to the network.

The function returns the number of devices linked. This includes the host itself. Before a device joins the link, the value zero will be returned by the function when called from that device.

Activity 21.4

Modify *Multiplayer02* changing the

```
Print("Connected")  
to  
Print("Number of clients : "+Str(GetNetworkNumClients  
(netid)))
```

Load the program onto as many devices as you have available and check that the count displayed matches the number of devices linked.

Save your project.

GetNetworkServerID()

Not only is the network assigned an ID, but each device on that network is also assigned a client ID. The client ID assigned to the host machine can be discovered using the `GetNetworkServerID()` statement (see FIG-21.8).

FIG-21.8

`GetNetworkServerID()` integer `GetNetworkServerID` ((id))

where

id is an integer value giving the ID assigned to the network.

GetNetworkMyClientID()

To discover the client ID of the machine currently running the program code, you can use the `GetNetworkMyClientID()` statement (see FIG-21.9).

FIG-21.9

`GetNetworkMyClientID()` integer `GetNetworkMyClientID` ((id))

where

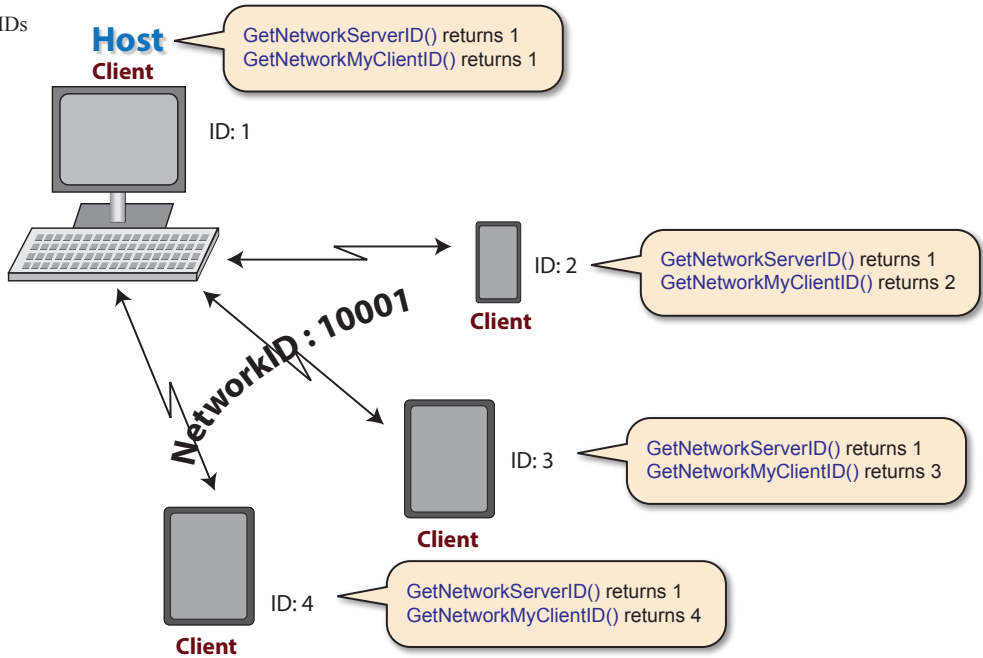
id is an integer value giving the ID assigned to the network.

The diagram in FIG-21.10 shows a network of four devices and the ID assigned to each device as well as the network ID. When exactly the same code is executed on

all four machines, the values returned by calls to `GetNetworkServerID()` and `GetNetworkMyClientID()` are shown.

FIG-21.10

Host and Client IDs



Activity 21.5

Modify *Multiplayer02* so that the host's ID and the current machine's client ID are displayed. Test your code.

Modify the program again so that the message *This is the host machine* is displayed only on the host machine. Test and save your program.

GetNetworkClientName()

The name assigned to any device on the network can be discovered using the `GetNetworkClientName()` statement (see FIG-21.11).

FIG-21.11

`GetNetworkClientName()` string `GetNetworkClientName` ((`id` , `idclient`))

where

- id** is an integer value giving the ID assigned to the network by the host.
- idclient** is an integer value giving the ID of the client whose name is to be returned.

Activity 21.6

Modify *Multiplayer02* so that the name of each device in the network is displayed. Test and save your code.

SetNetworkNoMoreClients()

It is likely that most of the games you write will only cater for a limited number of players at any one time, so we need to control the maximum number of devices that can join a particular session.

We can impose a limit on the number of devices linked by executing the `SetNetworkNoMoreClients()` statement from the host device. The statement has the format shown in FIG-21.12.

FIG-21.12

SetNetworkNoMoreClients()

`SetNetworkNoMoreClients ((id))`

where

id is an integer value giving the ID assigned to the network.

As you can see in the description, this is a statement that should only be executed by the host machine. But if all the machines in our network are executing exactly the same code, how are we to create statements which are executed only by the host? The simple way to do this is to start with the following condition:

```
if GetNetworkServerID(netid) = GetNetworkMyClientID(netid)
```

This condition can only be true when executed on the host machine, so to limit the number of clients to, say 2, we would use the lines

```
if GetNetworkServerID(netid) = GetNetworkMyClientID(netid)
  if GetNetworkNumClients(netid) = 2
    SetNetworkNoMoreClients(netid)
  endif
endif
```

Activity 21.7

Modify *Multiplayer02* so that no more than two clients are allowed (host + one other). *The necessary code must be added inside the if joined = 1..endif structure.*

If you have three devices available, test your program by attempting to add all three devices to the network.

The third device cannot join the network, but can it detect the network's ID and the host's ID? Save you project.

CloseNetwork()

Of course you can just close down the app to disconnect a device from the network, but the correct way to achieve disconnection is to get the machine to execute a `CloseNetwork()` statement. When executed on the host machine, this statement will shut down the complete network, but when executed from a client, only that client is disconnected. The statement has the format shown in FIG-21.13.

FIG-21.13

CloseNetwork()

`CloseNetwork ((id))`

where

id is an integer value giving the ID assigned to the network.

GetNetworkClientDisconnected()

The host maintains a list of the machines connected to it. When a device disconnects from the network, that list needs to be updated. The first step in updating the list is to check to see if a client has disconnected. This is done using the `GetNetworkClientDisconnected()` statement (see FIG-21.14) which returns 1 if a specified client has disconnected from the network.

FIG-21.14

GetNetworkClient
↳ Disconnected()

integer `GetNetworkClientDisconnected` ((`id` , `idclient`))

where

id is an integer value giving the ID assigned to the network.

idclient is an integer value giving the client device's ID.

DeleteNetworkClient()

When a client has disconnected, it needs to be removed from the list maintained by the host machine. This is done using the `DeleteNetworkClient()` statement (see FIG-21.15).

FIG-21.15

DeleteNetworkClient()

`DeleteNetworkClient` ((`id` , `idclient`))

where

id is an integer value giving the ID assigned to the network.

idclient is an integer value giving the ID of the client to be removed from the list.

GetNetworkFirstClient()

If we want the host's list of currently connected clients to remain up to date, then we need to implement code which performs the equivalent of:

```
FOR each client ID DO
  IF GetNetworkClientDisconnected(networkID, clientID) = 1
    DeleteNetworkClient(networkID, clientID)
  ENDIF
ENDFOR
```

The only problem we have is identifying the client IDs. From the displays we have seen in previous Activities, we can see that these IDs seem to start at 1 and increment by 1. So we have client IDs of 1, 2, 3 etc. However, this is hardly a foolproof way of finding out the client IDs.

Luckily, AGK provides us with two statements for identifying client IDs. The first of these is `GetNetworkFirstClient()` which returns the ID of the first client held in the host's list of clients. This statement has the format shown in FIG-21.16.

FIG-21.16

GetNetworkFirstClient()

integer `GetNetworkFirstClient` ((`id`))

where

id is an integer value giving the ID assigned to the network.

The function returns the ID of the first client in the list maintained by the host. If there are no clients, zero is returned.

GetNetworkNextClient()

Having found the ID of the first client, we can discover the ID of subsequent clients using the `GetNetworkNextClient()` statement which returns the next client in that list.

FIG-21.17

The statement has the format shown in FIG-21.17.

`GetNetworkNextClient()` integer `GetNetworkNextClient` (`id`)

where

id is an integer value giving the ID assigned to the network.

The function will return zero if all IDs have already been returned.

To work our way through the complete list of clients, we would use the code:

```
clientID = GetNetworkFirstClient(netid)
while clientID <> 0
    rem --- do something with client details here ---
    GetNetworkNextClient(netid)
endwhile
```

The program in FIG-21.18 displays a list of the current client IDs on the host machine. The ID of any client disconnecting from the network is removed from the list.

FIG-21.18

Managing Network
Clients

```
rem *** Joining and Leaving a Network ***
global netid
global joined = 0

SetUpButtons()
JoinTheNetwork()
do
    HandleButtons()
    rem *** Get Host to maintain a list of devices ***
    if joined = 1
        if GetNetworkServerID(netid) = GetNetworkMyClientID(netid)
            MaintainHostList()
        endif
    endif
    Sync()
loop

function SetUpButtons()
    rem *** Create Host, Client and Exit buttons ***
    AddVirtualButton(1,10,20,10)
    SetVirtualButtonText(1,"Host")
    AddVirtualButton(2,30,20,10)
    SetVirtualButtonText(2,"Join")
    AddVirtualButton(3,50,20,10)
    SetVirtualButtonText(3,"Exit")
endfunction
```



FIG-21.18

(continued)

Managing Network
Clients

```

function JoinTheNetwork()
    rem *** Wait for button press ***
    repeat
        HandleButtons()
        Sync()
    until joined = 1
endfunction

function HandleButtons()
    rem *** If Host button pressed and not already joined ***
    if GetVirtualButtonPressed(1)= 1 and joined = 0
        rem *** Host the app ***
        netid = HostNetwork("MyNetwork","Hostmachine",1026)
        rem *** Record as joined ***
        joined = 1
    endif
    rem *** If Client button pressed and not already joined ***
    if GetVirtualButtonPressed(2)=1 and joined = 0
        rem *** Join as client ***
        clientname$ = Str(GetSeconds())
        netid = JoinNetwork("MyNetwork",clientname$)
        joined = 1
        rem *** Record as joined ***
    endif
    rem *** If Exit button pressed, leave network ***
    if GetVirtualButtonPressed(3)=1 and joined = 1
        CloseNetwork(netid)
        joined = 0
    endif
endfunction

function MaintainHostList()
    rem *** Start with empty list ***
    list$=""
    rem *** Get first clientID ***
    clientid = GetNetworkFirstClient(netid)
    rem *** Deal with all the clients ***
    while clientid <> 0
        rem *** If this device has disconnected, display message
        ↵and remove for list ***
        if GetNetworkClientDisconnected(netid,clientid)=1
            Print("Disconnected "+str(clientid))
            DeleteNetworkClient(netid,clientid)
        else
            rem *** Add a connected client to the list ***
            list$ = list$ +Str(clientid)+" "
        endif
        rem *** Get the next client ***
        clientid = GetNetworkNextClient(netid)
    endwhile
    rem *** Display ID of all clients and number of clients ***
    Print(list$+" Number of clients : "+
    ↵Str(GetNetworkNumClients(netid)))
endfunction

```

Activity 21.8

Start a new project called *MultiPlayer03* and implement the code given in FIG-21.18.

Run the AGK on as many devices as you have available and broadcast the program to the devices.

Make your PC the host and have the other devices join as clients. *You should see a list of client IDs and the total number of clients displayed only on the PC.*

Press the **Exit** button on each client (not the host). *You may have to press more than once for the press to be detected. The list of clients should reduce as each device leaves the network.*

Run the setup again. This time have the clients leave the network by simply closing down the app (don't press the **Exit** button). The clients should disconnect just as before.

Save your project.

CreateNetworkMessage()

A common requirement of multiplayer games is the need to send information from one device to another. One method of achieving this is to send a message over the network. Constructing a message requires several steps, the first of these being the creation of the empty message structure. This is done using `CreateNetworkMessage()` (see FIG-21.19).

FIG-21.19

`CreateNetworkMessage()` integer `CreateNetworkMessage()` ()

This statement returns an ID assigned to the new message structure.

AddNetworkMessageFloat(), AddNetworkMessageInteger() and AddNetworkMessageString()

Once a message has been created, we can add data to the message using the `AddNetworkMessageFloat()`, `AddNetworkMessageInteger()`, or `AddNetworkMessageString()` as appropriate. The format for each of these statements is shown in FIG-21.20.

FIG-21.20

`AddMessageValueFloat()` `AddMessageValueInteger()` `AddMessageValueString()`

`AddMessageValueFloat()` (idmsg , fval)

`AddMessageValueInteger()` (idmsg , ival)

`AddMessageValueString()` (idmsg , sval)

where

idmsg is an integer value giving the ID of the message to which the value is to be added.

fval	is the real value to be added to the message.
ival	is the integer value to be added to the message.
sval	is the string value to be added to the message.

You are free to add as many values as you wish to a message. For example, let's assume we wish to send the screen coordinates of sprite 1 to other devices on the network, we would construct that message using the following commands:

```
messageid = CreateNetworkMessage()
AddMessageValueFloat(messageid, GetSpriteX(1))
AddMessageValueFloat(messageid, GetSpriteY(1))
```

SendNetworkMessage()

With the complete message prepared, we now need to send it to a specific client or to all other clients. This is done using the `SendNetworkMessage()` statement (see FIG-21.21).

FIG-21.21

SendNetworkMessage()

where

id	is an integer value giving the ID assigned to the network.
idclient	is an integer value giving the ID of the client to which the message is to be sent. If a value of zero is used, the message will be sent to all clients (excluding the client sending the message).
idmsg	is an integer value giving the ID of the message to which the value is to be added.

Once a message has been sent, that message resource is deleted from the client sending the message and the ID value it was assigned can be reused.

GetNetworkMessage()

Any client expecting to receive a message must execute the `GetNetworkMessage()` statement in order to gain access to that message.

The statement returns the ID of any message that has been received; if no message has been received, zero is returned.

FIG-21.22

The `GetNetworkMessage()` statement has the format shown in FIG-21.22.

GetNetworkMessage() integer

where

id	is an integer value giving the ID assigned to the network.
-----------	--

The function returns the ID of the message. This will be used to retrieve the data from the message.

GetNetworkMessageFloat(), GetNetworkMessageInteger() and GetNetworkMessageString()

When a message has been received, the data it contains can be extracted using the `GetNetworkMessageFloat()`, `GetNetworkMessageInteger()` and `GetNetworkMessageString()` statements (see FIG-21.23).

FIG-21.23

`float` `GetNetworkMessageFloat ((idmsg))`
`GetNetworkMessageFloat()`
`integer` `GetNetworkMessageInteger ((idmsg))`
`GetNetworkMessageInteger()`
`string` `GetNetworkMessageString ((idmsg))`
`GetNetworkMessageString()`

where

idmsg is an integer value giving the ID assigned to the message by a previously executed `GetNetworkMessage()` statement.

DeleteNetworkMessage()

Once all the required data has been extracted from a message, the message should be deleted by the receiving client using the `DeleteNetworkMessage()` statement (see FIG-21.24).

FIG-21.24

`DeleteNetworkMessage ((idmsg))`
`DeleteNetworkMessage()`

where

idmsg is an integer value giving the ID of the message to be deleted.

Since several messages may be received, it is best to handle them in a loop structure of the form:

```
rem *** Get the message ***
messageId = GetNetworkMessage(netid)
rem *** Deal with all messages received ***
while messageId <> 0
  rem --- Extract data from message at this point ---
  rem *** Delete message ***
  DeleteNetworkMessage(messageId)
  rem *** Get next message ***
  messageId = GetNetworkMessage(netid)
endwhile
```

The program in FIG-21.25 sets up a sprite on the host machine and then allows the user on a second machine to move that sprite around the host's screen. When the screen of the client machine is being touched (or a mouse button pressed), that client machine sends a message to the host giving the coordinates of the pointer. The host machine receives the message and uses the coordinates in the message to position the sprite.

FIG-21.25

Using Messages

```
rem *** Sending Messages over a Network ***
global netid
global joined = 0
```



FIG-21.25

(continued)

Using Messages

```

SetUpButtons()
JoinTheNetwork()
do
    rem *** If machine has joined the network ***
    if joined = 1
        rem *** If it's the host, move the sprite ***
        if GetNetworkServerID(netid) = GetNetworkMyClientID(netid)
            MoveHostSprite()
        else
            rem *** If it's the client send the pointer
            ↵coordinates ***
            SendPointerPosition()
        endif
    endif
    Sync()
loop

function SetUpButtons()
    rem *** Create Host and Client buttons ***
    AddVirtualButton(1,10,20,10)
    SetVirtualButtonText(1,"Host")
    AddVirtualButton(2,30,20,10)
    SetVirtualButtonText(2,"Join")
endfunction

function JoinTheNetwork()
    rem *** Wait for button press ***
    repeat
        HandleButtons()
        Sync()
    until joined = 1
    rem *** Delete all buttons ***
    DeleteVirtualButton(1)
    DeleteVirtualButton(2)
endfunction

function HandleButtons()
    rem *** If Host button pressed and not already joined ***
    if GetVirtualButtonPressed(1)= 1 and joined = 0
        rem *** Host the app ***
        netid = HostNetwork("MyNetwork","Hostmachine",1026)
        rem *** Create the sprite to be displayed ***
        CreateHostSprite()
        rem *** Record as joined ***
        joined = 1
    endif
    rem *** If Client button pressed and not already joined ***
    if GetVirtualButtonPressed(2)=1 and joined = 0
        rem *** Join as client ***
        clientname$ = Str(GetSeconds())
        netid = JoinNetwork("MyNetwork",clientname$)
        joined = 1
        rem *** Record as joined ***
    endif
endfunction

```



FIG-21.25

(continued)

Using Messages

```

function SendPointerPosition()
    rem *** If the pointer is pressed ***
    if GetPointerState() = 1
        rem *** Send message with coordinates ***
        messageId = CreateNetworkMessage()
        AddNetworkMessageFloat(messageId, GetPointerX())
        AddNetworkMessageFloat(messageId, GetPointerY())
        SendNetworkMessage(netid, GetNetworkServerID(netid),
            messageId)
    endif
endfunction

function CreateHostSprite()
    rem *** Create grey background ***
    SetClearColor(180,180,180)
    rem *** Create sprite ***
    LoadImage(1,"Crosshairs.png")
    CreateSprite(1,1)
    SetSpriteSize(1,10,-1)
endfunction

function MoveHostSprite()
    rem *** Get the message ***
    messageId = GetNetworkMessage(netid)
    rem *** Deal with all messages received ***
    while messageId <> 0
        rem *** Extract coordinates ***
        x# = GetNetworkMessageFloat(messageId)
        y# = GetNetworkMessageFloat(messageId)
        rem *** Move sprite ***
        SetSpritePosition(1,x#,y#)
        rem *** Delete message ***
        DeleteNetworkMessage(messageId)
        rem *** Get next message ***
        messageId = GetNetworkMessage(netid)
    endwhile
endfunction

```

Activity 21.9

Start a new project called *Multiplayer04*. Copy *Crosshairs.png* from *AGKDownloads/Chapter21/* into the project's *media* folder.

Implement the code given in FIG-21.25.

Create the PC as host and a second device as client. Touch the client's screen (or drag its mouse) and watch the host's sprite move in response.

Save your project.

GetNetworkMessageFromClient()

A machine which receives a message can discover the sender's ID using the `GetNetworkMessageFromClient()` statement (see FIG-21.26).

FIG-21.26

GetNetworkMessage
↳ FromClient()

integer `GetNetworkMessageFromClient` () idmsg ()

where

idmsg is an integer value giving the ID of the message whose sender's ID is to be returned.

Activity 21.10

In this Activity we are going to modify Multiplayer04 so that the host displays two sprites, each controlled from different clients.

Copy *Sphere.png* from *AGKDownloads/Chapter21/* into the project's *media* folder.

Modify the code for `CreateHostSprite()` to read

```
function CreateHostSprite()  
    rem *** Create grey background ***  
    SetClearColor(180,180,180)  
    rem *** Create sprite ***  
    LoadImage(1,"Crosshairs.png")  
    CreateSprite(1,1)  
    SetSpriteSize(1,10,-1)  
    LoadImage(2,"Sphere.png")  
    CreateSprite(2,2)  
    SetSpriteSize(2,10,-1)  
endfunction
```

In the `MoveHostSprite()` function, modify the code so that sprite 1 is moved only if the ID of the client sending the message is 2; otherwise move sprite 2.

Run the program on three devices and try moving the sprites using the two client machines.

Save your project.

SetNetworkLocalFloat() and SetNetworkLocalInteger()

We have already seen that a client can send messages to other clients, but messages are sender-initiated and the receiving client needs to handle the message.

A more informal way of handling data sharing between clients is to set up local real or integer variables using the `SetNetworkLocalFloat()` and `SetNetworkLocalInteger()` statements, which can then be read by other clients when required.

The variables set up in this way are given explicit names and other clients can refer to these named variables to discover their contents.

FIG-21.27

The formats of the `SetNetworkLocalFloat()` and `SetNetworkLocalInteger()` statements are shown in FIG-21.27.

SetNetworkLocalFloat()

`SetNetworkLocalFloat (id , svar , fval [, ireset])`

SetNetworkLocalInteger()

`SetNetworkLocalInteger (id , svar , ival [, ireset])`

where

id	is an integer value giving the ID assigned to the network.
svar	is a string giving the name of the variable to be used.
fval	is the real value to be assigned to the variable.
ival	is the integer value to be assigned to the variable.
ireset	is an integer value (0 or 1) which determines if the variable in question is to be reset to zero after it has been read by a client (1) or have its contents remain unchanged after reading (0). NOTE: Even if the reset option is selected, the variable is reset to zero only for the client reading the variable; if that same variable is read by a different client, that variable's original contents will still be available.

GetNetworkClientFloat() and GetNetworkClientInteger()

For a client to read the contents of the variables set up by `SetNetworkLocalFloat()` and `SetNetworkLocalInteger()`, the `GetNetworkClientFloat()` and `GetNetworkClientInteger()` statements must be used. These statements have the format shown in FIG-21.28.

FIG-21.28

`GetNetworkClientFloat()` float `GetNetworkClientFloat` ((`id` , `idclient` , `svar`))

`GetNetworkClientInteger()` integer `GetNetworkClientInteger` ((`id` , `idclient` , `svar`))

where

id	is an integer value giving the ID assigned to the network.
idclient	is an integer value giving the ID of the client from which a value is to be read.
svar	is a string giving the name of the variable to be read.

The functions return the value held in the named variable. A client can read the contents of a variable that it has itself set up simply by specifying its own client ID number.

If the variable was set up using the reset option, once a client has read the contents of that variable, should the same client attempt another read of the same variable, zero will be returned (unless the client which set up the variable has subsequently assigned it a new value).

The program in FIG-21.29 is designed to run on three devices (although it will work on two). Each device displays its own sprite and that sprite can be moved by touching the screen or dragging the mouse. Each screen also displays the sprites belonging to the other devices.

Once a device has joined the network, there are two main functions which control the program. The first of these is `SetOwnSpriteDetails()` which reads the screen pointer position and records this in local variables `x` and `y`. The second is `MoveAllSprites()`

which reads the contents of the variables *x* and *y* from each client and moves the sprites accordingly.

FIG-21.29

Using Local Variables

```
rem *** Using a Device's Local Values ***

global netid
global joined = 0

SetUpButtons()
JoinTheNetwork()
CreateSprites()
do
    SetOwnSpriteDetails()
    MoveAllSprites()
    Sync()
loop

function SetUpButtons()
    rem *** Create Host and Client buttons ***
    AddVirtualButton(1,10,20,10)
    SetVirtualButtonText(1,"Host")
    AddVirtualButton(2,30,20,10)
    SetVirtualButtonText(2,"Join")
endfunction

function JoinTheNetwork()
    rem *** Wait for button press ***
    repeat
        HandleButtons()
        Sync()
    until joined = 1
    rem *** Delete all buttons ***
    DeleteVirtualButton(1)
    DeleteVirtualButton(2)
endfunction

function HandleButtons()
    rem *** If Host button pressed and not already joined ***
    if GetVirtualButtonPressed(1)=1 and joined = 0
        rem *** Host the app ***
        netid = HostNetwork("MyNetwork","Hostmachine",1026)
        rem *** Record as joined ***
        joined = 1
    endif
    rem *** If Client button pressed and not already joined ***
    if GetVirtualButtonPressed(2)=1 and joined = 0
        rem *** Join as client ***
        clientname$ = Str(GetSeconds())
        netid = JoinNetwork("MyNetwork",clientname$)
        joined = 1
        rem *** Record as joined ***
    endif
endfunction
```



FIG-21.29

(continued)

Using Local Variables

```

function CreateSprites()
    rem *** Create grey background ***
    SetClearColor(180,180,180)
    rem *** Create sprites ***
    LoadImage(1,"Square.png")
    CreateSprite(1,1)
    SetSpriteSize(1,10,-1)
    LoadImage(2,"Sphere.png")
    CreateSprite(2,2)
    SetSpriteSize(2,10,-1)
    LoadImage(3,"Triangle.png")
    CreateSprite(3,3)
    SetSpriteSize(3,10,-1)
endfunction

function SetOwnSpriteDetails()
    rem *** Get pointer position ***
    if GetPointerState() = 1
        x# = GetPointerX()
        y# = GetPointerY()
        rem *** Make position available for all devices ***
        SetNetworkLocalFloat(netid,"x",x#)
        SetNetworkLocalFloat(netid,"y",y#)
    endif
endfunction

function MoveAllSprites()
    rem *** Get first device ID ***
    clientid = GetNetworkFirstClient(netid)
    rem *** Deal with all the clients ***
    while clientid <> 0
        rem *** Get coordinates from device ***
        x# = GetNetworkClientFloat(netid,clientid,"x")
        y# = GetNetworkClientFloat(netid,clientid,"y")
        rem *** Position device's sprite ***
        SetSpritePosition(clientid,x#,y#)
        rem *** Get the next client ***
        clientid = GetNetworkNextClient(netid)
    endwhile
endfunction

```

Activity 21.11

Start a new project called *Multiplayer05*. Copy the files *Sphere.png*, *Triangle.png* and *Square.png* from *AGKDownloads/Chapter21/* into the project's *media* folder.

Implement the code given in FIG-21.29.

Create the PC as host and two other devices as clients.

Drag the sprite on each device and watch it move on all three machines.

Save your project.

SetNetworkClientUserData()

If you want to store values unique to each client, then one option is use the client's data area. This is a 5-slot integer data area created specifically for the client device. The information in this data area cannot be read by other clients. To place a value in one of these slots, the `SetNetworkClientUserData()` statement is used (see FIG-21.30).

FIG-21.30 SetNetworkClientUserData()

`SetNetworkClientUserData ((id , idclient , indx , ival))`

where

- id** is an integer value giving the ID assigned to the network.
- idclient** is an integer value giving the ID assigned to the client. This must be the ID of the client running the code.
- indx** is an integer value (0 to 4) giving the slot to be assigned a value.
- ival** is the integer value to be assigned to the slot.

For example, the statement

```
SetNetworkClientUserData (netid,1,0,20)
```

would store the value 20 in slot zero of the host machine (always client 1).

GetNetworkClientUserData()

To retrieve a value held in a client's data area, the `GetNetworkClientUserData()` statement is used (see FIG-21.31).

FIG-21.31 GetNetworkClientUserData()

integer `GetNetworkClientUserData ((id , idclient , indx))`

where

- id** is an integer value giving the ID assigned to the network.
- idclient** is an integer value giving the ID assigned to the client. This must be the ID of the client running the code.
- indx** is an integer value (0 to 4) giving the slot whose value is to be retrieved.

For example,

```
num = GetNetworkClientUserData (netid,1,0)
```

would retrieve the value stored in slot zero of the host machine.

It is worth noting that the same effect can be achieved by simply using any variable declared within your program code, since each variable is local to the machine on which it is running.

GetNetworkClientPing()

We can check the time it takes to send information from a client to the host using the `GetNetworkClientPing()` statement which has the format shown in FIG-21.32.

FIG-21.32

GetNetworkClientPing() float `GetNetworkClientPing` ((`id` , `idclient`))

where

id is an integer value giving the ID assigned to the network.

idclient is an integer value giving the ID of the client to be pinged.

To determine how long it takes to pass data from one client to another, we must sum the ping times for each client involved since all data passes through the host machine.

SetNetworkLatency()

In a computer network, the term **latency** is used when referring to the delay between messages on the network.

AGK has a latency setting of 15 milliseconds. This means it allows 15 milliseconds for data to travel between clients before attempting to retrieve more data. This will be fine if you have a fast Wi-Fi connection and little data is being transferred. However, for poor connections or large amounts of data, the latency time should be increased.

You can change the network latency setting using the `SetNetworkLatency()` statement (see FIG-21.33).

FIG-21.33

SetNetworkLatency() `SetNetworkLatency` ((`id` , `ilat`))

where

id is an integer value giving the ID assigned to the network.

ilat is an integer value giving the latency setting in milliseconds.

CreateBroadcastListener()

A broadcast is a message sent by one or more devices which can be received by all machines with access to the network. When you set up an AGK multiplayer game, the AGK software automatically transmits the network name and other details on port 45631. A device can listen in on these messages using `CreateBroadcastListener()` (see FIG-21.34).

FIG-21.34

CreateBroadcastListener() integer `CreateBroadcastListener` ((`iport`))

where

iport is an integer value giving the port to be listened to.

The function returns the ID assigned to the listener created.

GetBroadcastMessage()

Once a listener has been created, you can check if the listener has detected a message using the `GetBroadcastMessage()` statement which has the format shown in FIG-21.35.

FIG-21.35

`GetBroadcastMessage()` integer `GetBroadcastMessage` (`idListen`)

where

idListen is an integer value giving the ID of the listener.

The function returns the ID of any message detected. If no message is detected, then zero is returned. Once a message is detected, `GetNetworkMessageInteger()`, `GetNetworkMessageFloat()` and `GetNetworkMessageString()` can be used to extract the contents of the message.

GetNetworkMessageFromIP()

The IP address of the device sending a message which has been received by a listener can be determined using the `GetNetworkMessageFromIP()` statement (see FIG-21.36).

FIG-21.36

`GetNetworkMessageFromIP()` string `GetNetworkMessageFromIP` (`idmsg`)

where

idmsg is an integer value giving the ID of the message whose sender's IP address is to be returned.

DeleteBroadcastListener()

When a listener is no longer required, it can be deleted and resources freed up using the `DeleteBroadcastListener()` statement (see FIG-21.37).

FIG-21.37

`DeleteBroadcastListener()` `DeleteBroadcastListener` (`idListen`)

where

idListen is an integer value giving the ID of the listener to be deleted.

The program in FIG-21.38 gives a simple example of a listener and how data is extracted from the received message.

FIG-21.38

Using a Listener

```
rem *** Using a Listener ***
rem *** Create text object ***
CreateText(1,"")
SetTextSize(1,3)
rem *** Create listener ***
listenId = CreateBroadcastListener(45631)
do
    rem *** IF broadcast received ***
    messID = GetBroadcastMessage(listenId)
    if messID <> 0
        rem *** Extract data from message and display it ***
```



FIG-21.38

(continued)

Using a Listener

```

        message$ = GetNetworkMessageString(messID)
        SetTextString(1,message$)
    endif
    Sync()
loop

```

Activity 21.12

Start a new project called *Listening* and implement the code given in FIG-21.38.

Hit the **Compile, Run and Broadcast** button.

What messages are displayed?

Save your project.

AGK broadcasts a message whenever an app is broadcast.

The text displayed is the default AGK network name, but if you set up your own network, then the name of that network is automatically broadcast. The program in FIG-21.39 hosts a network called *MyNetwork*, a name which will then automatically be broadcast by AGK.

FIG-21.39

Broadcasting the Network

```

rem *** Broadcasting the Network Name ***

rem *** Host the app ***
netid = HostNetwork("MyNetwork3","Hostmachine",1026)
Sync()

do
loop

```

Activity 21.13

Create a new project called *Broadcasting* and implement the code in FIG-21.39.

Save your project.

Load the AGK Player on your tablet or phone. Load *Listening* (created in Activity 21.12) and broadcast this to your tablet/phone.

Load *Broadcasting* and run it on your PC. What text appears on the tablet/phone?

Modify the `SetTextString()` statement in *Listening* to read:

```

SetTextString(1,message$+Chr(10)+"at IP address "+
↳GetNetworkMessageFromIP(messID))

```

Save *Listening* and broadcast it to your device then rerun *Broadcasting* from your PC.

Summary

- A Wi-Fi transmission is normally required to set up a local network.
- A network requires one machine to act as host.
- All machines added to a network (including the host) are clients.
- The network has its own unique name and ID.
- Every client on the network has its own unique name and ID.
- Use `HostNetwork()` to set up the network and host machine.
- All data on a network is transmitted via the host.
- The host maintains a list of all clients connected to the network.
- Use `IsNetworkActive()` to determine if a network of a specified ID is active.
- Use `JoinNetwork()` to join the network as a client machine.
- Use `GetNetworkNumClients()` to discover the number of clients a network has (this includes the host machine).
- Use `SetNetworkNoMoreClients()` to stop any new clients joining a network.
- Use `GetNetworkServerID()` to find the ID of the host machine.
- Use `GetNetworkMyClient()` to find the ID of the machine on which the app is running.
- Use `GetNetworkClientName()` to discover the name assigned to a specific client.
- Use `closeNetwork()` to close a specified network (when run from the host) or to disconnect from a network (when run from a client).
- Use `GetNetworkClientDisconnected()` to determine if a specified client has disconnected from the network.
- Use `DeleteNetworkClient()` to remove a disconnected client from the list of clients maintained by the host machine.
- Use `GetNetworkFirstClient()` to discover the ID of the first machine in the host's list of clients. This will normally be the ID of the host machine).
- Use `GetNetworkNextClient()` to determine the ID of the next client in the host's list of clients.
- Data can be transmitted between clients using a message.
- Each message has a unique ID.
- Use `CreateNetworkMessage()` to create an empty message.
- Use `AddNetworkMessageFloat()`, `AddNetworkMessageInteger()`, or `AddNetworkMessageString()` to add data to a message.
- Use `SendNetworkMessage()` to transmit a message to a specific client or to all clients.
- Use `GetNetworkMessage()` to retrieve the ID of a message received by a client.

- Use `GetNetworkMessageFloat()`, `GetNetworkMessageInteger()`, OR `GetNetworkMessageString()` to retrieve data from a message.
- Use `DeleteNetworkMessage()` to delete a received message.
- Use `GetNetworkMessageFromClient()` to discover the ID of the client which sent a message.
- Use `SetNetworkLocalFloat()` and `SetNetworkLocalInteger()` to assign values to local variables that can be accessed by other clients.
- Use `GetNetworkLocalFloat()` and `GetNetworkLocalInteger()` to access the local values set up by other clients.
- Use `SetNetworkClientUserData()` to assign a value to the client-specific 5 element integer data block.
- Use `GetNetworkClientUserData()` to retrieve a value from the client-specific 5 element integer data block.
- Use `GetNetworkClientPing()` to determine the time taken for a message to travel from a client to the host machine.
- Use `SetNetworkLatency()` to set the delay between data transmissions over the network.
- A broadcast is data which can be accessed by any device, even those that have not yet joined a network.
- Use `CreateBroadcasterListener()` to create a listener capable of picking up broadcasts.
- Use `GetBroadcastMessage()` to determine if a broadcast has been made.
- When a broadcast has been detected, the `GetNetworkMessage` functions can be used to retrieve the contents of the broadcast.
- `GetNetworkMessageFromIP()` to get the IP address of any device sending a message or broadcast.
- Use `DeleteBroadcastListener()` to destroy a listener which is no longer required.

Multi-Player Tic Tac Toe

Introduction

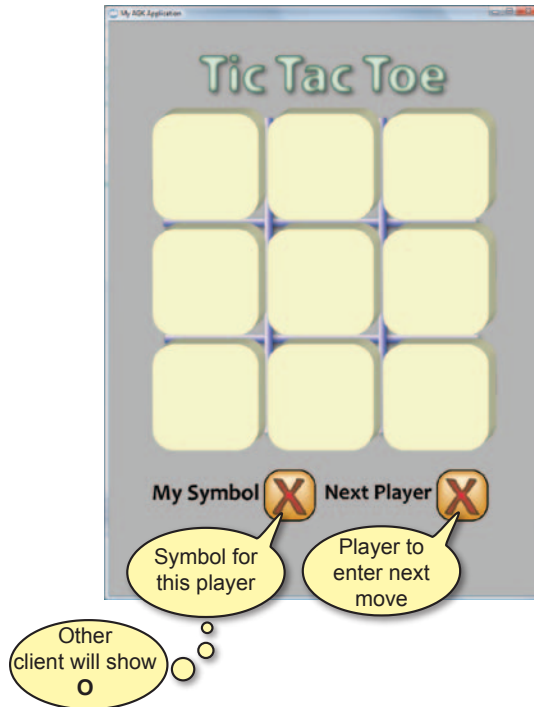
The game of Tic Tac Toe (called Noughts and Crosses in the UK) must be familiar to almost everyone on the planet, but just in case there are a few of you out there who have never played ...

Tic Tac Toe has a play area divided into 9 squares (3 by 3). Two players take turns at entering an X or an O (one symbol for each player). The first to get three of their symbols in a horizontal, vertical or diagonal row is the winner.

The layout of the app is shown in FIG-21.40.

FIG-21.40

TicTacToe Layout



Game Logic

The app makes use of the following logic:

```
Set up network connections
Show game layout
Player is X
REPEAT
  IF it's your turn THEN
    Enter your move
  ELSE
    Get other player's move
  ENDIF
  Update the screen
  Player is next player
UNTIL game complete
Show end screen
```

Global Variables

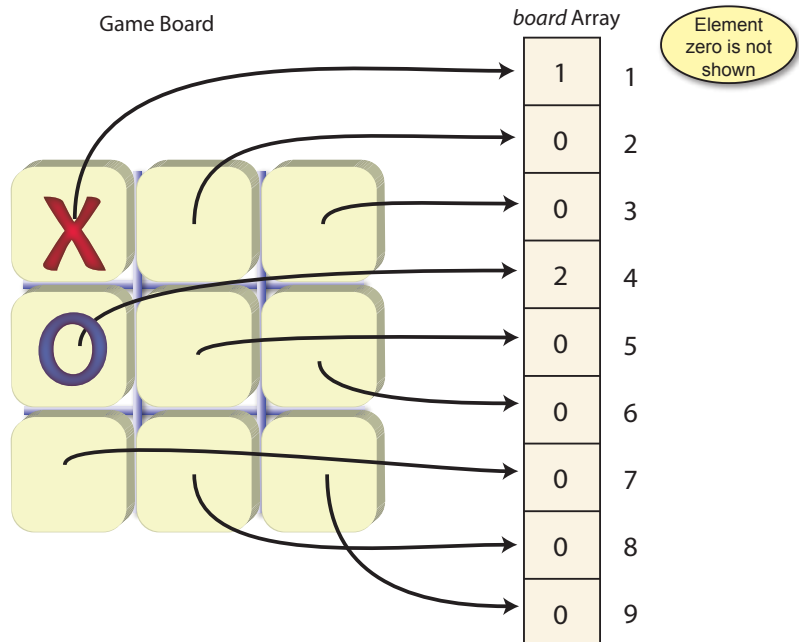
As usual, there are a few global variables needed by the program. These are:

```
rem *** Global Variables ***
global dim board[9]=[0,0,0,0,0,0,0,0,0] //The playing area
↳ (0:empty,1:X,2:O)
global player                                //Current player
↳ (1:X,2:O)
global joined=0                               //Joined network
↳ (0:no,1:yes)
global netid                                  //Network ID
global clientid                               //This client's ID
global movesmade                             //Number of moves made
```

The *board* array is used to represent the state of the game, with one cell of the array representing one square on the board. How the squares map to the array is shown in FIG-21.41.

FIG-21.41

How the Game Layout
Maps to the Array



As you can see from FIG-21.41, an empty square on the board is represented by a zero in the corresponding cell of the array, while an X is represented by a 1 and an O by a 2.

Main Program Logic

The main logic of the program reflects the structured English logic given earlier:

```
SetupNetwork()
SetUpBoard()
rem *** X to play ***
player = 1
repeat
    rem *** IF this client's turn ***
```

```

        if player = clientid
            rem *** Enter move ***
            move = MakeMove()
        else
            rem *** IF other client's turn, get their move ***
            move = GetMove()
        endif
        rem *** Update screen with new move ***
        UpdateScreen(move)
        rem *** Change player to move ***
        player = 3 - player
        rem *** Check to see if game is finished ***
        r = IsComplete()
        Sync()
    until r<>0
    rem *** Show winning line or game drawn message ***
    FinishGame(r mod 10)
    Sync()
end

```

Although the parameters to the various function calls may not yet be clear, we can easily see the overall logic from this code. Perhaps the most unusual line is:

```
player = 3 - player
```

The player variable is set to 1 if it is X's turn to play and 2 if it is O's turn. The line of code used ensures that the variable player switches between these two values each time the line is executed (3-1 is 2 and 3-2 is 1).

Activity 21.14

Start a new project called *TicTacToe* and implement the code given so far.

Add test stubs for each of the functions called. Use *square* as the parameter for `UpdateScreen()` and *status* as the parameter for `FinishGame()`. Test and save your project.

SetUpNetwork()

This function connects the two machines to the network using **Host** and **Join** buttons. It will shut down the app if there are not two machines in the network within 30 seconds. The code for the function is:

```

function SetUpNetwork()
    rem *** Create Host and Client buttons ***
    AddVirtualButton(1,10,20,10)
    SetVirtualButtonText(1,"Host")
    AddVirtualButton(2,30,20,10)
    SetVirtualButtonText(2,"Join")
    rem *** Wait till button pressed ***
    repeat
        HandleButtons()
        Sync()
    until joined = 1
    rem *** Wait until both machines connected or timed out (30
    ↵seconds)***
    time = GetSeconds()
    repeat
    until GetNetworkNumClients(netid)=2 or GetSeconds()-time>30
    rem *** If we don't have 2 clients, close game ***

```

```

    if GetNetworkNumClients(netid)<>2
        CloseGame()
    endif
    rem *** Get client's id ***
    clientid = GetNetworkMyClientID(netid)
    rem *** Delete the network buttons ***
    DeleteVirtualButton(1)
    DeleteVirtualButton(2)
endfunction

```

As you can see, this function calls to helper functions, `HandleButtons()` and `CloseGame()`. These are coded as:

```

function HandleButtons()
    rem *** If Host button pressed and not already joined ***
    if GetVirtualButtonPressed(1)= 1 and joined = 0
        rem *** Host the app ***
        netid = HostNetwork("TicTacToe","Hostmachine",1026)
        rem *** Record as joined ***
        joined = 1
    endif
    rem *** If Client button pressed and not already joined ***
    if GetVirtualButtonPressed(2)=1 and joined = 0
        rem *** Join as client ***
        clientname$ = Str(GetSeconds())
        netid = JoinNetwork("TicTacToe",clientname$)
        rem *** Record as joined ***
        joined = 1
    endif
endfunction

```

```

function CloseGame()
    rem *** Display message ***
    CreateText(1,"Game needs two machines to play")
    SetTextSize(1,3)
    SetTextPosition(1,10,47)
    Sync()
    rem *** Wait 5 seconds then close game ***
    Sleep(5000)
end
endfunction

```

Activity 21.15

Add the new functions to *TicTacToe* and identifying comments to your program. Compile and save your project.

SetUpBoard()

This function displays the visuals of the game and also creates a set of hidden sprites - one over each of the nine squares of the board. It is these hidden sprites which will be used to detect the position selected by the player.

The function's code is:

```

function SetUpBoard()
    rem *** Set aspect ratio ***
    SetDisplayAspect(768/1024.0)

```

```

SetClearColor(180,180,180)
Sync()
rem *** Load Images ***
LoadImage(1,"TTTX.png")
LoadImage(2,"TTTO.png")
LoadImage(3,"TTTTitle.png")
LoadImage(4,"TTTBoard.png")
LoadImage(5,"TTTMySymbol.png")
LoadImage(6,"TTTNxtPlay.png")
LoadImage(7,"TTTtile01.png")
rem *** Title ***
CreateSprite(3,3)
SetSpriteSize(3,60,-1)
SetSpritePosition(3,20,5)
rem *** Board ***
CreateSprite(4,4)
SetSpriteSize(4,80,-1)
SetSpritePosition(4,10,15)
rem *** My Symbol Legend ***
CreateSprite(5,5)
SetSpriteSize(5,25,-1)
SetSpritePosition(5,10,80)
rem *** Symbol Box ***
CreateSprite(7,7)
SetSpriteSize(7,12,-1)
SetSpritePosition(7,36,78)
rem *** Player Shape ***
CreateSprite(1,clientid)
SetSpriteSize(1,8,-1)
SetSpritePosition(1,38,79.5)
rem *** Next Player Legend ***
CreateSprite(6,6)
SetSpriteSize(6,25,-1)
SetSpritePosition(6,50,80)
rem *** Symbol Box ***
CreateSprite(8,7)
SetSpriteSize(8,12,-1)
SetSpritePosition(8,76,78)
rem *** Next Player Shape ***
CreateSprite(9,1)
SetSpriteSize(9,8,-1)
SetSpritePosition(9,78,79.5)
rem *** Both symbols (X and O) in sprite ***
SetSpriteAnimation(9,153,164,1)
AddSpriteAnimationFrame(9,2)
rem *** Create a set of hidden sprites ***
rem *** one over each of the 9 areas on board ***
id = 100
for row = 0 to 2
  for col = 0 to 2
    inc id
    CreateSprite(id,0)
    SetSpriteDepth(id,9)
    SetSpriteSize(id,25,-1)
    SetSpritePosition(id,10+col*27,16.5+row*20)
    SetSpriteVisible(id,0)
  next col
next row
Sync()
endfunction

```


Activity 21.16

From *AGKDownloads/Chapter21*, copy the necessary image files used by `SetUpBoard()` into the project's *media* folder.

Add the code for `SetUpBoard()` to your program.

Start up the *AGKPlayer* in your second device and broadcast the program.

Create a host and client, then check that the game board appears on both devices. Check that one device shows X and the other O beside the *My Symbol* text.

Save your project.

MakeMove()

The `MakeMove()` function detects a press on an empty square on the board, records the move in the *board* array and sends a network message of the move made. The function also returns the move made as a value between 1 and 9 (the square's number in the *board* array).

The function's code is:

```
function MakeMove()
    rem *** Get empty square selected by player ***
    state = 0
    repeat
        repeat
            if GetPointerState() = 1 and state = 0
                id = GetSpriteHit(GetPointerX(),GetPointerY())
                state= 1
            else
                state = 0
            endif
            Sync()
        until id >= 101 and id <= 109
        square = id - 100
    until board[square]=0
    rem *** Record player's move in array ***
    board[square] = player
    rem *** Send message of move made to other device ***
    messageID = CreateNetworkMessage()
    AddNetworkMessageInteger(messageID,player)
    AddNetworkMessageInteger(messageID,square)
    SendNetworkMessage(netid,3-clientid,messageID)
endfunction square
```

GetMove()

When it is the opponent's turn to move, then the details of that move are retrieved over the network using the `GetMove()` function.

```
function GetMove()
    rem *** Get message from network ***
    repeat
        messageID = GetNetworkMessage(netid)
    until messageID <> 0
```

```

player = GetNetworkMessageInteger (messageID)
square = GetNetworkMessageInteger (messageID)
rem *** Update board array ***
board[square] = player
rem *** Delete the message ***
DeleteNetworkMessage (messageID)
endfunction square

```

UpdateScreen()

This function updates the appearance of the screen, placing an X or O symbol at the position of the latest move, changing the symbol of the next player and incrementing the number of moves made.

The code is:

```

function UpdateScreen(square)
    rem *** Create the sprite to be placed on board ***
    spriteid = 20+square
    CreateSprite(spriteid,player)
    SetSpriteSize(spriteid,12,-1)
    rem *** Position sprite ***
    SetSpritePosition(spriteid,17+(spriteid-21) mod 3*27,
    ↵ 21+(spriteid-21)/3*20)
    rem *** Change Next Player symbol ***
    SetSpriteFrame(9,3-player)
    Sync()
    rem *** Add 1 to the number of moves made ***
    inc movesmade
endfunction

```

Activity 21.17

Add the `MakeMove()`, `GetMove()` and `UpdateScreen()` functions to your program.

Start up the *AGKPlayer* in your second device and broadcast the program.

Check that you can now play the game over the two devices.

Save your project.

IsComplete()

The *IsComplete()* function returns two pieces of information stored in a single variable.

The first value determines if the game is complete. This value is 0 if the game is not complete; if the game has finished, it is a number between 1 and 9. The value determines where on the board the winning line has occurred (1: top horizontal, 2: middle horizontal, 3: bottom horizontal, 4: left vertical, 5: middle vertical, 6: right vertical, 7 diagonal top-left to bottom-right, 8: diagonal bottom-left to top-right. The final value, 9, is used to indicate a draw.

The second value is the winner of the game (0: no winner, 1: X wins, 2: O wins).

To return both pieces of information in a single value, the *winner* value is multiplied

by 10 and added to the *game-complete* value. For example, if O wins with a line on the left vertical, then the function would return the value 24. The two values can be extracted from this returned value using the following formulae:

```
winner      = returned-value / 10
game-complete = returned-value mod 10
```

Activity 21.18

Create the code for the `IsComplete()` function and add it to your project.

To test the function, get the program to display the value returned, splitting it into the original two values.

Save your project.

FinishGame()

The game finishes by either displaying a *GAME DRAWN* message or by drawing a line through the winning three squares. Both of these options are actually performed by other routines.

The code for `FinishGame()` is:

```
function FinishGame(status)
    rem *** Draw or win ***
    if status = 9
        GameDrawn()
    else
        ShowWinningLine(status)
    endif
endfunction
```

GameDrawn()

`GameDrawn()` displays *GAME DRAWN* at the centre of the board for 5 seconds.

The routine's code is:

```
function GameDrawn()
    rem *** Load image used ***
    LoadImage(99,"TTTDraw.png")
    rem *** Display message ***
    CreateSprite(99,99)
    SetSpriteSize(99,60,-1)
    SetSpritePositionByOffset(99,50,47)
    Sync()
    rem *** Wait 5 seconds ***
    Sleep(5000)
endfunction
```

ShowWinningLine()

This function draws a line through the winning squares. The position of the line is determined by the function's parameter.

The function contains the following code:

```

function ShowWinningLine(line)
    rem *** Load image used ***
    LoadImage (99,"TTTLine.png")

    rem *** Create sprite ***
    CreateSprite(99,99)
    SetSpriteSize(99,-1,60)
    rem *** Place horizontally ...***
    if line <= 2
        SetSpriteAngle(99,90)
        SetSpritePositionByOffset(99,49,26+line*20)
    endif
    rem *** Place vertically ...***
    if line >= 3 and line <= 5
        SetSpritePosition(99,20+(line-3)*27,16)
    endif
    rem *** Diagonal TL to BR ***
    if line = 6
        SetSpriteAngle(99,135)
        SetSpritePositionByOffset(99,49,47)
    endif
    rem *** Diagonal BL to TR ***
    if line = 7
        SetSpriteAngle(99,45)
        SetSpritePositionByOffset(99,49,47)
    endif
    Sync()
    rem *** Wait for 5 seconds ***
    Sleep(5000)
endfunction

```

Activity 21.19

From *AGKDownloads/Chapter21*, copy the necessary image files used by `GameDrawn()` and `ShowWinningLine()` into the project's *media* folder.

Add the code for `IsComplete()`, `GameDrawn()` and `ShowWinningLine()` to your program.

Start up the *AGKPlayer* in your second device and broadcast the program.

Check that the game is now complete.

Save your project.

Introduction

The letters HTTP stand for **H**yper**T**ext **T**ransfer **P**rotocol.

Hypertext is the term used for text which has clickable links to other text. It's the sort of stuff you use all the time when you're on the internet clicking links to take you from one page to another.

Transfer Protocol refers to the agreed format for transferring data between devices.

In an HTTP environment, the **client** machine makes a request for data and the **host** (also known as the **server**) fulfills that request by sending the appropriate file.

A variation is the HTTPS connection. The extra S stands for Security. The data transferred in this way is encrypted for greater security.

When you use your browser software to connect to the Internet, you are sending and receiving data using HTTP. If you purchase items from an online store, then your details will be encrypted using HTTPS.

HTTP Statements

OpenBrowser()

The simplest way to make an HTTP connection is simply to run your browser software. You can do this from within a program using `OpenBrowser()` (see FIG-21.42).

FIG-21.42

OpenBrowser()

`OpenBrowser ((surl))`

where

surl is a string giving the URL (web address) to be opened when the browser is loaded.

After opening your browser the app will continue to execute separately on some devices.

The program in FIG-21.43 loads your device's Internet browser when you press the **Browse** button and then displays an incrementing count.

FIG-21.43

Loading a Browser from an App

```
Rem *** Use Internet Browser ***

rem *** Create browse button ***
LoadImage(1,"BrowseButton.png")
CreateSprite(1,1)
SetSpriteSize(1,20,-1)
SetSpritePositionByOffset(1,50,50)

rem *** Start count at zero ***
count = 0
```



FIG-21.43

(continued)

Loading a Browser from
an App

```

do
    rem *** Display count ***
    Print(count)
    Sync()
    rem *** Add 1 to count ***
    inc count
    rem If button pressed, start browser ***
    if GetSpriteHit(GetPointerX(),GetPointerY()) = 1 and
    ↵GetPointerPressed() = 1
        OpenBrowser("www.digital-skills.co.uk")
    endif
loop

```

Activity 21.20

Start a new project called *UseInternet* and implement the code given in FIG-21.43. Copy the file *AGKDownloads/Chapter21/BrowseButton.png* to the *media* folder.

Run the program, press the **Browse** button and check how the displayed count goes on increasing while the browser is running. Save your project.

CreateHTTPConnection()

The first stage in the file transfer request is to set up a connection resource. This is done using the `CreateHTTPConnection()` statement (see FIG-21.44).

FIG-21.44

CreateHTTPConnection()

integer **CreateHTTPConnection** (())

The function returns an integer giving the ID assigned to the HTTP resource.

A typical statement would be:

```
id = CreateHTTPConnection()
```

SetHTTPHost()

Once the HTTP resource has been created, you need to specify the domain on the host machine along with any username and password that may be required. This is done using the `SetHTTPHost()` statement (see FIG-21.45).

FIG-21.45 SetHTTPHost()

integer **SetHTTPHost** ((id , sdomain , itype [, suser , spass]))

where

id is an integer value giving the ID previously assigned to the connection.

sdomain is a string giving the domain on the server that is to be accessed.

itype is an integer (0 or 1) giving the type of connection (0: HTTP, 1: HTTPS).

user is a string giving the username required for site access.

spass is a string giving the user's password.

The function returns 1 if a successful connection is made, otherwise zero is returned.

To access a standard public website, we would use a command such as:

```
SetHTTPHost(id,"www.digital-skills.co.uk",0)
```

GetHTTPFile()

Once your app has connected to a domain, you can download a file from that site using the `GetHTTPFile()` statement (see FIG-21.46).

FIG-21.46

GetHTTPFile()

integer `GetHTTPFile` (`(` `id` `,` `sfile` `,` `save` [`,` `spost`] `)`

where

id is an integer value previously assigned to the connection.

sfile is a string giving the name of the file on the server which is to be downloaded.

save is a string giving the name of the file where the downloaded data is to be saved on the client.

spost is a string giving any data that the client wishes to send to the server.

The function returns 1 if the specified file was located and transfer started, otherwise zero is returned.

To download the typical homepage file of a website and save it in a file called *saved.html*, we could use a line such as:

```
success = GetHTTPFile(id,"index.html","saved.html")
```

The transfer of a large file may take some time and continues in the background while subsequent statements in your program continue to be executed.

GetHTTPFileProgress()

To check on the progress of a file download initiated by a `GetHTTPFile()` command, use `GetHTTPFileProgress()` (see FIG-21.47).

FIG-21.47

GetHTTPFileProgress()

float `GetHTTPFileProgress` (`(` `id` `)`

where

This function can also be used to check on the progress of file uploads - as we will see later.

id is an integer value previously assigned to the connection.

The function returns a real value in the range 0 to 100 which indicates (not always accurately) what percentage of the file has been downloaded.

For example, we could display the progress of the download with a statement such

as:

```
Print(GetHTTPFileProgress(id))
```

GetHTTPFileComplete()

Even if `GetHTTPProgress()` returns a value of 100, that is not a guarantee that the download is complete. The only way to be sure that all the data has safely arrived is to call `GetHTTPFileComplete()` (see FIG-21.48).

FIG-21.48

`GetHTTPFileComplete()`

integer `GetHTTPFileComplete` ((id))

where

id is an integer value previously assigned to the connection.

The function returns 1 if the transfer is complete, otherwise zero is returned.

CloseHTTPConnection()

When you no longer require an HTTP connection to a specific host, you can disconnect using `CloseHTTPConnection()` (see FIG-21.49).

FIG-21.49

`CloseHTTPConnection()`

`CloseHTTPConnection` ((id))

where

id is an integer value previously assigned to the connection.

Once disconnected, you can reattach to a different host by making a new call to `SetHTTPHost()`.

DeleteHTTPConnection()

Once an HTTP connection has been closed and is no longer required, then you can delete the resource using `DeleteHTTPConnection()` (see FIG-21.50).

FIG-21.50

`DeleteHTTPConnection()`

`DeleteHTTPConnection` ((id))

where

id is an integer value previously assigned to the connection.

A complete example of how to download a public accessible file is shown in FIG-21.51.

FIG-21.51

How to Download a File
Using HTTP

```
rem *** Download a file from a website ***  
  
rem *** Create HTTP connection Resource ***  
id = CreateHTTPConnection()  
  
rem *** Specify unsecured web address ***  
result = SetHTTPHost(id,"www.digital-skills.co.uk",0)
```



FIG-21.51

(continued)

How to Download a File
Using HTTP

```

rem *** If found try to download a file ***
if result = 1
    rem *** If file found download it ***
    if GetHTTPFile(id,"AGK-Ch14.pdf","AGK14-TextResources.pdf")=1
        rem *** Wait until download complete ***
        repeat
            rem *** Print progress ***
            Print(GetHTTPFileProgress(id))
            Sync()
        until GetHTTPFileComplete(id)=1
    else
        rem *** Error if file not found ***
        Print("File not found")
        Sync()
    endif
    rem *** Close the connection ***
    CloseHTTPConnection(id)
else
    rem *** Error if address not found ***
    Print("Not connected")
    Sync()
endif

rem *** Delete HTTP resource ***
DeleteHTTPConnection(id)

rem *** Display end message ***
Print("Program Completed")
do
loop

```

Activity 21.21

Start a new project called *Download01* and implement the code in FIG-21.51.

Run and save your project.

Go to the folder used by your project and check that the file has been correctly downloaded.

SendHTTPRequest()

A second option when retrieving a file is to have the contents of that file returned as a single string. You can do this by using the `SendHTTPRequest()` statement (see FIG-21.52).

FIG-21.52

SendHTTPRequest()

string `SendHTTPRequest` (`id` , `sfile` [`, spost`])

where

- id** is an integer value previously assigned to the connection.
- sfile** is a string giving the name of the file on the server which is to be downloaded.
- spost** is a string giving any additional data that needs to be sent to the server.

The string returned by the function represents the contents of the file. If the file cannot be accessed, an empty string is returned.

After executing `CreateHTTPConnection()` and `SetHTTPHost()`, a typical `SendHTTPRequest()` statement would be:

```
result$ = SendHTTPRequest(id,"index.html")
```

Unlike, `GetHTTPFile()` which allows the download to happen in the background while your app continues to execute, `SendHTTPRequest()` will halt your program until the contents of the file have been loaded into the specified string variable.

The code in FIG-21.53 gives a simple example of how the statement can be used.

FIG-21.53

Using
`SendHTTPRequest()`

```
rem *** Download a file from a website ***

rem *** Create HTTP connection Resource ***
id = CreateHTTPConnection()

rem *** Specify unsecured web address ***
result = SetHTTPHost(id,"www.digital-skills.co.uk",0)

rem *** Get file's contents ***
r$ = SendHTTPRequest(id,"index.html")

rem *** Display the file ***
Print(r$)
Sync()

do
loop
```

Activity 21.22

Start a new project called *Download02* and implement the code in FIG-21.53.

Run and save your project. What output is produced?

SendHTTPRequestASync()

If you want the contents of a file returned as a string (as offered by `SendHTTPRequest()`), but you don't want the program to stop while the data is sent, you can begin a request for a download using `SendHTTPRequestASync()` (see FIG-21.54).

FIG-21.54

`SendHTTPRequest`
↳ `ASync()`

integer `SendHTTPRequestASync` (`(` `id` `,` `sfile` `[` `,` `spost` `]` `)`

where

id is an integer value previously assigned to the connection.

sfile is a string giving the name of the file on the server which is to be downloaded.

spost is a string giving any additional data that needs to be sent to the server.

GetHTTPResponseReady()

FIG-21.55

GetHTTPResponse
Ready()

To check if the server is ready to respond to the client's request to send data, use `GetHTTPResponseReady()` (see FIG-21.55).

integer `GetHTTPResponseReady()` (`id`)

where

`id` is an integer value previously assigned to the connection.

The function will return 1 if the server is ready to give a response to a previous `SendHTTPRequestASync()` call.

GetHTTPResponse()

FIG-21.56

GetHTTPResponse()

Once we know the server is ready to respond (having called `GetHTTPResponseReady()`), we can finally receive the contents of the requested file using `GetHTTPResponse()` (see FIG-21.56).

string `GetHTTPResponse()` (`id`)

where

`id` is an integer value previously assigned to the connection.

The function returns a string containing the contents of the requested file. If the file could not be accessed, an empty string is returned.

The program in FIG-21.57 demonstrates how `SendHTTPRequestASync()` is used. While the program awaits a response, it increments a displayed value to demonstrate the ability of the program to continue execution.

FIG-21.57

Using Asynchronous
Download

```
rem *** Download a file from a website 2 ***

rem *** Create HTTP connection Resource ***
id = CreateHTTPConnection()
rem *** Specify unsecured web address ***
result = SetHTTPHost(id,"www.digital-skills.co.uk",0)

rem *** Request a file ***
SendHTTPRequestASync(id,"AGK-Ch14.pdf")

rem *** Count until server ready ***
count = 0
repeat
    inc count
    Print(count)
    Sync()
until GetHTTPResponseReady(id)=1
rem *** Get the file ***
r$ = GetHTTPResponse(id)
rem *** Display the file ***
Print(r$)
Sync()
do
loop
```

Activity 21.23

Start a new project called *Download03* and implement the code in FIG-21.57.

Run and save your project.

SendHTTPFile()

As well as receiving files from a server, you may also want to upload files. This would allow your app to do things such as keep a list of the highest scores achieved by all users.

Of course, you need to persuade the server to accept an uploaded file. Normally, you would begin by creating a script file on the server that accepts and handles the file your app wants to upload.

Although it is outside the scope of this book to discuss the creation of the necessary script, a sample PHP file is shown in FIG-21.58.

FIG-21.58

PHP Script for Uploading
a File

Thanks to Paul Johnston
for supplying this script.

```
<?
    function fail( $msg ) { echo $msg; exit(); }

    //uploaded file
    if ( $_FILES["myfile"]["tmp_name"] == "" ) fail( "No file
    ↳uploaded." );

    if ( $_FILES["myfile"]["error"] > 0 )
    {
        switch ( $_FILES['myfile']['error'] )
        {
            case 1: fail( 'File exceeded maximum server upload
            ↳size.' ); break;
            case 2: fail( 'File exceeded maximum file size.' );
            ↳break;
            case 3: //partial file
            case 4: fail( 'An error occurred during file upload,
            ↳please try again' ); break;    //no file
        }
    }

    $filename = basename( $_FILES["myfile"]['name'] );
    $size = (int) $_FILES["myfile"]["size"];

    //check file does not exist and place it in its new location
    $filepath = "gamesfolder/$filename.dat";

    if ( file_exists($filepath) ) fail("That file has already been
    ↳uploaded. If this should not be the case please contact
    ↳support quoting the filename.");
    if ( @move_uploaded_file( $_FILES["myfile"]["tmp_name"],
    ↳$filepath ) == FALSE ) fail("Could not move uploaded file,
    ↳please try again.");
    if ( @chmod( $filepath, 0744 ) == FALSE ) fail("Could not
    ↳modify uploaded file, please contact support.");

?>
```

This script will accept the uploaded file and copy it to a folder named *gamesfolder*.

To upload a file to the server use `SendHTTPFile()` which has the format shown in FIG-21.59.

FIG-21.59

`SendHTTPFile()`

integer `SendHTTPFile` ((`id` , `script` , `spost` , `sfile`)

where

id	is an integer value previously assigned to the connection.
script	is a string specifying the script file to be run on the server. The string may include path details if the script is not in the same folder as your <i>index.html</i> file.
spost	is a string giving any additional data that needs to be sent to the server.
sfile	is a string giving the name of the file to be sent to the server. If the file is not in the <i>media</i> folder, then relative path details need to be included.

If the script expects to read the value of one or more variables, then the names and values of the variables would be given in the *spost* parameter. For example, if we wanted to specify that variables *myvar* and *desc* have the values 5.6 and *test* respectively, the value of the *spost* parameter would be “*myvar=5.6&desc=test*”.

If a variable contains a string value, any non-alphanumeric characters such as a space, comma, question mark, etc. must be encoded using a % symbol followed by the required character’s ASCII code given in hexadecimal. For example, if you wanted to pass the string “*Is this correct?*” in a variable called *myvar2*, the *spost* parameter would be written as

```
"myvar2=Is%20this%20correct%3F"
```

In the PHP script, variables *myvar* and *desc* would be accessed using the terms

```
$_POST['myvar']
```

and

```
$_POST['desc']
```

Activity 21.24

To perform this activity you will need to have your own website on which you can store and run PHP files.

Create a file called *upload.php* containing the text shown in FIG-21.58.

Use your website’s file manager to upload the file to the main directory of your website.

Using your website’s file manager, create a sub-folder called *gamesfolder* off your main directory.

Your server will almost certainly use a Unix-based setup. Since Unix is a case-sensitive system, it is important that you make sure the file and folder names match exactly throughout.

When you use `SendHTTPFile()`, the upload takes place as a background event, allowing your program to continue execution. We can use `GetHTTPFileProgress()` to discover what percentage of the file has been uploaded and `GetHTTPResponseReady()` to check when the upload is complete.

If a problem has occurred (such as the file being too large, or a file of that name already exists) then `GetHTTPResponse()` will return the appropriate message generated by the script.

The program in FIG-21.60 uploads a file called *scores.dat* to *www.digital-skills.co.uk* in a subfolder called *gamesfolder*.

FIG-21.60

Uploading a File

```
rem *** Upload a file to a website ***

rem *** Create HTTP connection Resource ***
id = CreateHTTPConnection()

rem *** Specify unsecured web address ***
result = SetHTTPHost(id,"www.digital-skills.co.uk",0)

rem *** Upload file ***
SendHTTPFile(id,"upload.php","", "scores.dat")
repeat
    Print(GetHTTPFileProgress(id))
    Sync()
until GetHTTPResponseReady(id)=1

rem *** Display script's response ***
Print(GetHTTPResponse(id))

rem *** Display end of program ***
Print("Transfer complete")
Sync()

do
loop
```

Activity 21.25

This activity follows on from Activity 21.24.

Create a project called *UploadFile* and implement the code given in FIG-21.60.

Run the program and then check the contents of *gamesfolder* on your web server and ensure that it now contains a file called *scores.dat*.

Summary

- HTTP stands for HyperText Transfer Protocol.
- HTTPS stands for HyperText Transfer Protocol Secure.
- Normally, communicating with a website requires the use of HTTP or HTTPS.
- Use `OpenBrowser()` to open your web browser from within an AGK app.
- Use `CreateHTTPConnection()` to set up an HTTP connection resource.

- Use `SetHTTPHost()` to specify the website with which you wish to communicate.
- Use `GetHTTPFile()` to start downloading a specified file from the website.
- Your app will continue to run while the file is being downloaded.
- Use `GetHTTPFileProgress()` to discover what percentage of the requested file has been downloaded.
- Use `GetHTTPFileComplete()` to check if the file download is complete.
- Use `CloseHTTPConnection()` to terminate the connection to the host.
- Use `DeleteHTTPConnection()` to delete the HTTP resource created earlier.
- Use `SendHTTPRequest()` to download the contents of a named file as a single string.
- When using `SendHTTPRequest()`, your app will halt until the download is complete.
- Use `SendHTTPRequestASync()` to download a file as a string while your program continues execution.
- Use `GetHTTPResponseReady()` to check that the server is ready to transmit the requested string.
- Use `GetHTTPResponse()` to retrieve the required string.
- Use `SendHTTPFile()` to upload a file to the server.
- When uploading a file to the server, make sure the appropriate script exists on the server to accept and handle the uploaded file.

Solutions

Activity 21.1

The network ID on my setup was 100001.

Activity 21.2

The message on the second device changes to *Not connected* shortly after the host machine's app is closed.

Activity 21.3

No solution required.

Activity 21.4

No solution required.

Activity 21.5

Modified code for *Multiplayer02*:

```
rem *** Testing Multiplayer Statements ***
rem *** Create Host/Join buttons ***
AddVirtualButton(1,10,20,10)
SetVirtualButtonText(1,"Host")
AddVirtualButton(2,30,20,10)
SetVirtualButtonText(2,"Join")
rem *** Not yet joined the network ***
joined = 0
do
    rem *** If Host button pressed and not already
    ⤵joined ***
    if GetVirtualButtonPressed(1)= 1 and joined = 0
        rem *** Host the app ***
        netid = HostNetwork("MyNetwork",
            ⤵"Hostmachine",1026)
        rem *** Record as joined ***
        joined = 1
    endif
    rem *** If Join button pressed and not already
    ⤵joined ***
    if GetVirtualButtonPressed(2)=1 and joined = 0
        rem *** Join as client ***
        netid = JoinNetwork("MyNetwork",
            ⤵Str(GetSeconds()))
        rem *** Record as joined ***
        joined = 1
    endif
    rem *** If joined, check network is active ***
    if joined = 1
        if IsNetworkActive(netid)
            Print("Number of clients : "+
                ⤵Str(GetNetworkNumClients(netid)))
        else
            Print("Not connected")
        endif
        Print("Host ID : "+
            ⤵Str(GetNetworkServerID(netid)))
        Print("This client's ID : "+
            ⤵Str(GetNetworkMyClientID(netid)))
        if GetNetworkServerID(netid) =
            ⤵GetNetworkMyClientID(netid)
            Print("This is the host machine")
        endif
    endif
    rem *** Display the network's ID ***
    Print("Network ID : "+Str(netid))
    Sync()
loop
```

Activity 21.6

In *Multiplayer02*, the `if joined =1..endif` structure's code is changed to

```
if joined = 1
    if IsNetworkActive(netid)
        Print("Number of clients : "+
            ⤵Str(GetNetworkNumClients(netid)))
```

```
    else
        Print("Not connected")
    endif
    Print("Host ID : "+
        ⤵Str(GetNetworkServerID(netid)))
    Print("This client's ID : "+
        ⤵Str(GetNetworkMyClientID(netid)))
    Print("This client's name : "+
        ⤵GetNetworkClientName(netid,
            ⤵GetNetworkMyClientID(netid)))
    if GetNetworkServerID(netid) =
        ⤵GetNetworkMyClientID(netid)
        Print("This is the host machine")
    endif
endif
```

Activity 21.7

In *Multiplayer02*, the `if joined =1..endif` structure's code is changed to:

```
if joined = 1
    if IsNetworkActive(netid)
        Print("Number of clients : "+
            ⤵Str(GetNetworkNumClients(netid)))
    else
        Print("Not connected")
    endif
    Print("Host ID : "+
        ⤵Str(GetNetworkServerID(netid)))
    Print("This client's ID : "+
        ⤵Str(GetNetworkMyClientID(netid)))
    Print("This client's name : "+
        ⤵GetNetworkClientName(netid,
            ⤵GetNetworkMyClientID(netid)))
    if GetNetworkServerID(netid) =
        ⤵GetNetworkMyClientID(netid)
        Print("This is the host machine")
        if GetNetworkNumClients(netid) = 2
            SetNetworkNoMoreClients(netid)
        endif
    endif
endif
```

Even although only two devices may be part of the network, other devices can detect the network's ID and the host's ID.

Activity 21.8

No solution required.

Activity 21.9

No solution required.

Activity 21.10

The modified code for *MoveHostSprite()*:

```
function MoveHostSprite()
    rem *** Get the message ***
    messageId = GetNetworkMessage(netid)
    rem *** Deal with all messages received ***
    while messageId <> 0
        rem *** Extract coordinates ***
        x# = GetNetworkMessageFloat(messageId)
        y# = GetNetworkMessageFloat(messageId)
        rem *** Move sprite ***
        if GetNetworkMessageFromClient(messageId)=2
            SetSpritePosition(1,x#,y#)
        else
            SetSpritePosition(2,x#,y#)
        endif
        rem *** Delete message ***
        DeleteNetworkMessage(messageId)
        rem *** Get next message ***
        messageId = GetNetworkMessage(netid)
    endwhile
endfunction
```

Activity 21.11

No solution required.

Activity 21.12

Two messages are displayed:

App Viewer Network

and

App Control Network

Activity 21.13

The message displayed is:

MyNetwork3

The IP address should also appear when the `Print()` statement is modified.

Activity 21.14

Code for *TicTacToe*:

```

rem *** Global Variables ***
global dim board[9]=[0,0,0,0,0,0,0,0,0] //The
  playing area (0:empty,1:X,2:O)
global player //Current
  player (1:X,2:O)
global joined=0 //Joined
  network (0:no,1:yes)
global netid //Network ID
global clientid //This
  client's ID
global movesmade //Number of
  moves made

```

```

SetUpNetwork()
SetUpBoard()
rem *** X to play ***
player = 1
repeat
  rem *** IF this client's turn ***
  if player = clientid
    rem *** Enter move ***
    move = MakeMove()
  else
    rem *** IF other client's turn, get
    their move ***
    move = GetMove()
  endif
  rem *** Update screen with new move ***
  UpdateScreen(move)
  rem *** Change player to move ***
  player = 3 - player
  rem *** Check to see if game is finished ***
  r = IsComplete()
  Sync()
until r<>0
rem *** Show winning line or game drawn message
***
FinishGame(r mod 10)
Sync()
end

```

```

function SetUpNetwork()
  Print("SetUpNetwork()")
endfunction

```

```

function SetUpBoard()
  Print("SetUpBoard()")
endfunction

```

```

function MakeMove()
  Print("MakeMove()")
endfunction 1

```

```

function GetMove()
  Print("GetMove()")
endfunction 1

```

```

function UpdateScreen(square)
  Print("UpdateScreen()")
endfunction

```

```

function IsComplete()

```

```

  Print("IsComplete()")
endfunction 0
function FinishGame(status)
  Print("FinishGame()")
endfunction

```

Activity 21.15

Modified code for *TicTacToe*:

```

rem *****
rem *** TicTacToe ***
rem *****

rem *** Global Variables ***
global dim board[9]=[0,0,0,0,0,0,0,0,0] //The
  playing area (0:empty,1:X,2:O)
global player //Current
  player (1:X,2:O)
global joined=0 //Joined
  network (0:no,1:yes)
global netid //Network ID
global clientid //This
  client's ID
global movesmade //Number of
  moves made

rem *****
rem *** Main Program Logic ***
rem *****

SetUpNetwork()
SetUpBoard()
rem *** X to play ***
player = 1
repeat
  rem *** IF this client's turn ***
  if player = clientid
    rem *** Enter move ***
    move = MakeMove()
  else
    rem *** IF other client's turn, get
    their move ***
    move = GetMove()
  endif
  rem *** Update screen with new move ***
  UpdateScreen(move)
  rem *** Change player to move ***
  player = 3 - player
  rem *** Check to see if game is finished ***
  r = IsComplete()
  Sync()
until r<>0
rem *** Show winning line or game drawn message
***
FinishGame(r mod 10)
Sync()
end

```

```

rem *****
rem *** Level 1 Functions ***
rem *****

```

```

function SetUpNetwork()
  rem *** Create Host and Client buttons ***
  AddVirtualButton(1,10,20,10)
  SetVirtualButtonText(1,"Host")
  AddVirtualButton(2,30,20,10)
  SetVirtualButtonText(2,"Join")
  rem *** Wait till button pressed ***
  repeat
    HandleButtons()
    Sync()
  until joined = 1
  rem *** Wait until both machines connected or
  timed out (30 seconds)***
  time = GetSeconds()
  repeat
    until GetNetworkNumClients(netid)=2 or
    GetSeconds()-time>30
  rem *** If we don't have 2 clients, close game
  ***
  if GetNetworkNumClients(netid)<>2
    CloseGame()
  endif
  rem *** Get client's id ***
  clientid = GetNetworkMyClientID(netid)

```

```

    rem *** Delete the network buttons ***
    DeleteVirtualButton(1)
    DeleteVirtualButton(2)
endfunction

function SetUpBoard()
    Print("SetUpBoard()")
endfunction

function MakeMove()
    Print("MakeMove()")
endfunction 1

function GetMove()
    Print("GetMove()")
endfunction 1

function UpdateScreen(square)
    Print("UpdateScreen()")
endfunction

function IsComplete()
    Print("IsComplete()")
endfunction 0

function FinishGame(status)
    Print("FinishGame()")
endfunction

rem *****
rem *** Level 2 Functions ***
rem *****

function HandleButtons()
    rem *** If Host button pressed and not already
    ↵joined ***
    if GetVirtualButtonPressed(1)= 1 and joined = 0
        rem *** Host the app ***
        netid = HostNetwork("TicTacToe",
        "Hostmachine",1026)
        rem *** Record as joined ***
        joined = 1
    endif
    rem *** If Client button pressed and not already
    ↵joined ***
    if GetVirtualButtonPressed(2)=1 and joined = 0
        rem *** Join as client ***
        clientname$ = Str(GetSeconds())
        netid = JoinNetwork("TicTacToe",clientname$)
        rem *** Record as joined ***
        joined = 1
    endif
endfunction

function CloseGame()
    rem *** Display message ***
    CreateText(1,"Game needs two machines to play")
    SetTextSize(1,3)
    SetTextPosition(1,10,47)
    Sync()
    rem *** Wait 5 seconds then close game ***
    Sleep(5000)
end
endfunction

```

Activity 21.16

Modified code for *TicTacToe*:

```

rem *****
rem *** TicTacToe ***
rem *****

rem *** Global Variables ***
global dim board[9]=[0,0,0,0,0,0,0,0,0] //The
↵playing area (0:empty,1:X,2:O)
global player //Current
↵player (1:X,2:O)
global joined=0 //Joined
↵network (0:no,1:yes)
global netid //Network ID
global clientid //This
↵client's ID
global movesmade //Number of
↵moves made

rem *****
rem *** Main Program Logic ***
rem *****

```

```

SetUpNetwork()
SetUpBoard()
rem *** X to play ***
player = 1
repeat
    rem *** IF this client's turn ***
    if player = clientid
        rem *** Enter move ***
        move = MakeMove()
    else
        rem *** IF other client's turn, get
        ↵their move ***
        move = GetMove()
    endif
    rem *** Update screen with new move ***
    UpdateScreen(move)
    rem *** Change player to move ***
    player = 3 - player
    rem *** Check to see if game is finished ***
    r = IsComplete()
    Sync()
until r<>0
rem *** Show winning line or game drawn message
↵***
FinishGame(r mod 10)
Sync()
end

```

```

rem *****
rem *** Level 1 Functions ***
rem *****

```

```

function SetUpNetwork()
    rem *** Create Host and Client buttons ***
    AddVirtualButton(1,10,20,10)
    SetVirtualButtonText(1,"Host")
    AddVirtualButton(2,30,20,10)
    SetVirtualButtonText(2,"Join")
    rem *** Wait till button pressed ***
    repeat
        HandleButtons()
        Sync()
    until joined = 1
    rem *** Wait until both machines connected or
    ↵timed out (30 seconds)***
    time = GetSeconds()
    repeat
        until GetNetworkNumClients(netid)=2 or
        ↵GetSeconds()-time>30
        rem *** If we don't have 2 clients, close game
        ↵***
        if GetNetworkNumClients(netid)<>2
            CloseGame()
        endif
        rem *** Get client's id ***
        clientid = GetNetworkMyClientID(netid)
        rem *** Delete the network buttons ***
        DeleteVirtualButton(1)
        DeleteVirtualButton(2)
    endfunction

```

```

function SetUpBoard()
    rem *** Set aspect ratio ***
    SetDisplayAspect(768/1024.0)
    SetClearColor(180,180,180)
    Sync()
    rem *** Load Images ***
    LoadImage(1,"TTTX.png")
    LoadImage(2,"TTTO.png")
    LoadImage(3,"TTTTitle.png")
    LoadImage(4,"TTTBoard.png")
    LoadImage(5,"TTTMySymbol.png")
    LoadImage(6,"TTTNxtPlay.png")
    LoadImage(7,"TTTtile01.png")
    rem *** Title ***
    CreateSprite(3,3)
    SetSpriteSize(3,60,-1)
    SetSpritePosition(3,20,5)
    rem *** Board ***
    CreateSprite(4,4)
    SetSpriteSize(4,80,-1)
    SetSpritePosition(4,10,15)
    rem *** My Symbol Legend ***
    CreateSprite(5,5)
    SetSpriteSize(5,25,-1)
    SetSpritePosition(5,10,80)
    rem *** Symbol Box ***
    CreateSprite(7,7)

```

```

SetSpriteSize(7,12,-1)
SetSpritePosition(7,36,78)
rem *** Player Shape ***
CreateSprite(1,clientid)
SetSpriteSize(1,8,-1)
SetSpritePosition(1,38,79.5)
rem *** Next Player Legend ***
CreateSprite(6,6)
SetSpriteSize(6,25,-1)
SetSpritePosition(6,50,80)
rem *** Symbol Box ***
CreateSprite(8,7)
SetSpriteSize(8,12,-1)
SetSpritePosition(8,76,78)
rem *** Next Player Shape ***
CreateSprite(9,1)
SetSpriteSize(9,8,-1)
SetSpritePosition(9,78,79.5)
rem *** Both symbols (X and O) in sprite ***
SetSpriteAnimation(9,153,164,1)
AddSpriteAnimationFrame(9,2)
rem *** Create a set of hidden sprites ***
rem *** one over each of the 9 areas on board
***
id = 100
for row = 0 to 2
  for col = 0 to 2
    inc id
    CreateSprite(id,0)
    SetSpriteDepth(id,9)
    SetSpriteSize(id,25,-1)
    SetSpritePosition(id,10+col*27,16.5+
      row*20)
    SetSpriteVisible(id,0)
  next col
next row
Sync()
endfunction

function MakeMove()
  Print("MakeMove()")
endfunction 1

function GetMove()
  Print("GetMove()")
endfunction 1

function UpdateScreen(square)
  Print("UpdateScreen()")
endfunction

function IsComplete()
  Print("IsComplete()")
endfunction 0

function FinishGame(status)
  Print("FinishGame()")
endfunction

rem *****
rem ***      Level 2 Functions      ***
rem *****

function HandleButtons()
  rem *** If Host button pressed and not already
  joined ***
  if GetVirtualButtonPressed(1) = 1 and joined = 0
    rem *** Host the app ***
    netid = HostNetwork("TicTakToe",
      "Hostmachine",1026)
    rem *** Record as joined ***
    joined = 1
  endif
  rem *** If Client button pressed and not already
  joined ***
  if GetVirtualButtonPressed(2)=1 and joined = 0
    rem *** Join as client ***
    clientname$ = Str(GetSeconds())
    netid = JoinNetwork("TicTakToe",clientname$)
    rem *** Record as joined ***
    joined = 1
  endif
endfunction

function CloseGame()
  rem *** Display message ***
  CreateText(1,"Game needs two machines to play")
  SetTextSize(1,3)
  SetTextPosition(1,10,47)

```

```

Sync()
rem *** Wait 5 seconds then close game ***
Sleep(5000)
end
endfunction

```

Activity 21.17

Modified code for *TicTacToe*:

```

rem *****
rem ***      TicTacToe      ***
rem *****

rem *** Global Variables ***
global dim board[9]=[0,0,0,0,0,0,0,0,0] //The
  playing area (0:empty,1:X,2:O)
global player //Current
  player (1:X,2:O)
global joined=0 //Joined
  network (0:no,1:yes)
global netid //Network ID
global clientid //This
  client's ID
global movesmade //Number of
  moves made

rem *****
rem ***      Main Program Logic      ***
rem *****

SetUpNetwork()
SetUpBoard()
rem *** X to play ***
player = 1
repeat
  rem *** IF this client's turn ***
  if player = clientid
    rem *** Enter move ***
    move = MakeMove()
  else
    rem *** IF other client's turn, get
    their move ***
    move = GetMove()
  endif
  rem *** Update screen with new move ***
  UpdateScreen(move)
  rem *** Change player to move ***
  player = 3 - player
  rem *** Check to see if game is finished ***
  r = IsComplete()
  Sync()
until r<>0
rem *** Show winning line or game drawn message
***
FinishGame(r mod 10)
Sync()
end

rem *****
rem ***      Level 1 Functions      ***
rem *****

function SetUpNetwork()
  rem *** Create Host and Client buttons ***
  AddVirtualButton(1,10,20,10)
  SetVirtualButtonText(1,"Host")
  AddVirtualButton(2,30,20,10)
  SetVirtualButtonText(2,"Join")
  rem *** Wait till button pressed ***
  repeat
    HandleButtons()
  until joined = 1
  rem *** Wait until both machines connected or
  timed out (30 seconds)***
  time = GetSeconds()
  repeat
    until GetNetworkNumClients(netid)=2 or
      GetSeconds()-time>30
  rem *** If we don't have 2 clients, close game
  ***
  if GetNetworkNumClients(netid)<2
    CloseGame()
  endif
  rem *** Get client's id ***
  clientid = GetNetworkMyClientID(netid)

```

```

    rem *** Delete the network buttons ***
    DeleteVirtualButton(1)
    DeleteVirtualButton(2)
endfunction

function SetUpBoard()
    rem *** Set aspect ratio ***
    SetDisplayAspect(768/1024.0)
    SetClearColor(180,180,180)
    Sync()
    rem *** Load Images ***
    LoadImage(1,"TTTX.png")
    LoadImage(2,"TTTO.png")
    LoadImage(3,"TTTTitle.png")
    LoadImage(4,"TTTBoard.png")
    LoadImage(5,"TTTMySymbol.png")
    LoadImage(6,"TTTNxtPlay.png")
    LoadImage(7,"TTTtile01.png")
    rem *** Title ***
    CreateSprite(3,3)
    SetSpriteSize(3,60,-1)
    SetSpritePosition(3,20,5)
    rem *** Board ***
    CreateSprite(4,4)
    SetSpriteSize(4,80,-1)
    SetSpritePosition(4,10,15)
    rem *** My Symbol Legend ***
    CreateSprite(5,5)
    SetSpriteSize(5,25,-1)
    SetSpritePosition(5,10,80)
    rem *** Symbol Box ***
    CreateSprite(7,7)
    SetSpriteSize(7,12,-1)
    SetSpritePosition(7,36,78)
    rem *** Player Shape ***
    CreateSprite(1,clientid)
    SetSpriteSize(1,8,-1)
    SetSpritePosition(1,38,79.5)
    rem *** Next Player Legend ***
    CreateSprite(6,6)
    SetSpriteSize(6,25,-1)
    SetSpritePosition(6,50,80)
    rem *** Symbol Box ***
    CreateSprite(8,7)
    SetSpriteSize(8,12,-1)
    SetSpritePosition(8,76,78)
    rem *** Next Player Shape ***
    CreateSprite(9,1)
    SetSpriteSize(9,8,-1)
    SetSpritePosition(9,78,79.5)
    rem *** Both symbols (X and O) in sprite ***
    SetSpriteAnimation(9,153,164,1)
    AddSpriteAnimationFrame(9,2)
    rem *** Create a set of hidden sprites ***
    rem *** one over each of the 9 areas on board ***
    id = 100
    for row = 0 to 2
        for col = 0 to 2
            inc id
            CreateSprite(id,0)
            SetSpriteDepth(id,9)
            SetSpriteSize(id,25,-1)
            SetSpritePosition(id,10+col*27,16.5+
                row*20)
            SetSpriteVisible(id,0)
        next col
    next row
    Sync()
endfunction

function MakeMove()
    rem *** Get empty square selected by player ***
    state = 0
    repeat
        if GetPointerState() = 1 and state = 0
            id = GetSpriteHit(GetPointerX(),
                GetPointerY())
            state = 1
        else
            state = 0
        endif
    until id >= 101 and id <= 109
    square = id - 100
    until board[square]=0
    rem *** Record player's move in array ***
    board[square] = player
    rem *** Send message of move made to other

```

```

    device ***
    messageID = CreateNetworkMessage()
    AddNetworkMessageInteger(messageID,player)
    AddNetworkMessageInteger(messageID,square)
    SendNetworkMessage(netid,3-clientid,messageID)
endfunction square

function GetMove()
    rem *** Get message from network ***
    repeat
        messageID = GetNetworkMessage(netid)
    until messageID <> 0
    player = GetNetworkMessageInteger(messageID)
    square = GetNetworkMessageInteger(messageID)
    rem *** Update board array ***
    board[square] = player
    rem *** Delete the message ***
    DeleteNetworkMessage(messageID)
endfunction square

function UpdateScreen(square)
    rem *** Create the sprite to be placed on board ***
    spriteid = 20+square
    CreateSprite(spriteid,player)
    SetSpriteSize(spriteid,12,-1)
    rem *** Position sprite ***
    SetSpritePosition(spriteid,17+(spriteid-21) mod
        3*27,21+(spriteid-21)/3*20)
    rem *** Change Next Player symbol ***
    SetSpriteFrame(9,3-player)
    Sync()
    rem *** Add 1 to the number of moves made ***
    inc movesmade
endfunction

function IsComplete()
    Print("IsComplete()")
endfunction 0

function FinishGame(status)
    Print("FinishGame()")
endfunction

rem *****
rem *** Level 2 Functions ***
rem *****

function HandleButtons()
    rem *** If Host button pressed and not already
    joined ***
    if GetVirtualButtonPressed(1) = 1 and joined = 0
        rem *** Host the app ***
        netid = HostNetwork("TicTacToe",
            "Hostmachine",1026)
        rem *** Record as joined ***
        joined = 1
    endif
    rem *** If Client button pressed and not already
    joined ***
    if GetVirtualButtonPressed(2)=1 and joined = 0
        rem *** Join as client ***
        clientname$ = Str(GetSeconds())
        netid = JoinNetwork("TicTacToe",clientname$)
        rem *** Record as joined ***
        joined = 1
    endif
endfunction

function CloseGame()
    rem *** Display message ***
    CreateText(1,"Game needs two machines to play")
    SetTextSize(1,3)
    SetTextPosition(1,10,47)
    Sync()
    rem *** Wait 5 seconds then close game ***
    Sleep(5000)
endfunction

```

Activity 21.18

One possible coding for *IsComplete()*:

```
function IsComplete()
```

```

winner = 0
line = 0
for c = 1 to 2
    rem *** Check rows ***
    for row = 1 to 7 step 3
        if board[row] = c and board[row+1] = c
            and board[row+2]=c
                winner = c
                line = (row-1) / 3
            endif
        next row
    rem *** Check columns ***
    for col = 1 to 3
        if board[col]=c and board[col+3]=c
            and board[col+6]=c
                winner = c
                line = 2+col
            endif
        next col
    rem *** Check diagonals ***
    if board[1]=c and board[5]=c
        and board[9]=c
            winner = c
            line = 6
        elseif board[3]=c and board[5]=c and
            board[7]=c
            winner = c
            line = 7
        endif
    next c
    result = winner*10+line
    rem *** If no winning line, check for a draw ***
    if result = 0 and movesmade = 9
        result = 9
    endif
endfunction result

```

To check the value returned by the `IsComplete()` function we could add the following lines immediately before the endfunction statement:

```

Print("Value returned : "+Str(result))
Print("Winner       : "+Str(result / 10))
Print("Line/Draw    : "+Str(result mod 10))
Sync()
Sleep(3000)

```

Activity 21.19

The complete code for *TicTacToe()*:

```

rem *****
rem ***          TicTacToe          ***
rem *****

rem *** Global Variables ***
global dim board[9]=[0,0,0,0,0,0,0,0,0] //The
    playing area (0:empty,1:X,2:O)
global player                                //Current
    player (1:X,2:O)
global joined=0                                //Joined
    network (0:no,1:yes)
global netid                                //Network ID
global clientid                            //This
    client's ID
global movesmade                            //Number of
    moves made

rem *****
rem ***          Main Program Logic          ***
rem *****

SetUpNetwork()
SetUpBoard()
rem *** X to play ***
player = 1
repeat
    rem *** IF this client's turn ***
    if player = clientid
        rem *** Enter move ***
        move = MakeMove()
    else
        rem *** IF other client's turn, get
            their move ***
        move = GetMove()
    endif
    rem *** Update screen with new move ***

```

```

UpdateScreen(move)
rem *** Change player to move ***
player = 3 - player
rem *** Check to see if game is finished ***
r = IsComplete()
Sync()
until r<>0
rem *** Show winning line or game drawn message
****
FinishGame(r mod 10)
Sync()

```

```

end
rem *****
rem ***          Level 1 Functions          ***
rem *****

```

```

function SetUpNetwork()
    rem *** Create Host and Client buttons ***
    AddVirtualButton(1,10,20,10)
    SetVirtualButtonText(1,"Host")
    AddVirtualButton(2,30,20,10)
    SetVirtualButtonText(2,"Join")
    rem *** Wait till button pressed ***
    repeat
        HandleButtons()
        Sync()
    until joined = 1
    rem *** Wait until both machines connected or
        timed out (30 seconds)***
    time = GetSeconds()
    repeat
        until GetNetworkNumClients(netid)=2 or
            GetSeconds()-time>30
        rem *** If we don't have 2 clients, close game
        ****
        if GetNetworkNumClients(netid)<>2
            CloseGame()
        endif
        rem *** Get client's id ***
        clientid = GetNetworkMyClientID(netid)
        rem *** Delete the network buttons ***
        DeleteVirtualButton(1)
        DeleteVirtualButton(2)
    endfunction

```

```

function SetUpBoard()
    rem *** Set aspect ratio ***
    SetDisplayAspect(768/1024.0)
    SetClearColor(180,180,180)
    Sync()
    rem *** Load Images ***
    LoadImage(1,"TTTX.png")
    LoadImage(2,"TTTO.png")
    LoadImage(3,"TTTTitle.png")
    LoadImage(4,"TTTBoard.png")
    LoadImage(5,"TTTMySymbol.png")
    LoadImage(6,"TTTNxtPlay.png")
    LoadImage(7,"TTTTile01.png")
    rem *** Title ***
    CreateSprite(3,3)
    SetSpriteSize(3,60,-1)
    SetSpritePosition(3,20,5)
    rem *** Board ***
    CreateSprite(4,4)
    SetSpriteSize(4,80,-1)
    SetSpritePosition(4,10,15)
    rem *** My Symbol Legend ***
    CreateSprite(5,5)
    SetSpriteSize(5,25,-1)
    SetSpritePosition(5,10,80)
    rem *** Symbol Box ***
    CreateSprite(7,7)
    SetSpriteSize(7,12,-1)
    SetSpritePosition(7,36,78)
    rem *** Player Shape ***
    CreateSprite(1,clientid)
    SetSpriteSize(1,8,-1)
    SetSpritePosition(1,38,79.5)
    rem *** Next Player Legend ***
    CreateSprite(6,6)
    SetSpriteSize(6,25,-1)
    SetSpritePosition(6,50,80)
    rem *** Symbol Box ***
    CreateSprite(8,7)
    SetSpriteSize(8,12,-1)
    SetSpritePosition(8,76,78)
    rem *** Next Player Shape ***
    CreateSprite(9,1)
    SetSpriteSize(9,8,-1)
    SetSpritePosition(9,78,79.5)

```

```

rem *** Both symbols (X and O) in sprite ***
SetSpriteAnimation(9,153,164,1)
AddSpriteAnimationFrame(9,2)
rem *** Create a set of hidden sprites ***
rem *** one over each of the 9 areas on board ***
id = 100
for row = 0 to 2
  for col = 0 to 2
    inc id
    CreateSprite(id,0)
    SetSpriteDepth(id,9)
    SetSpriteSize(id,25,-1)
    SetSpritePosition(id,10+col*27,16.5+row*20)
    SetSpriteVisible(id,0)
  next col
next row
Sync()
endfunction

function MakeMove()
  rem *** Get empty square selected by player ***
  state = 0
  repeat
    if GetPointerState() = 1 and state = 0
      id = GetSpriteHit(GetPointerX(),GetPointerY())
      state = 1
    else
      state = 0
    endif
  until id >= 101 and id <= 109
  square = id - 100
  until board[square]=0
  rem *** Record player's move in array ***
  board[square] = player
  rem *** Send message of move made to other device ***
  messageID = CreateNetworkMessage()
  AddNetworkMessageInteger(messageID,player)
  AddNetworkMessageInteger(messageID,square)
  SendNetworkMessage(netid,3-clientid,messageID)
endfunction square

function GetMove()
  rem *** Get message from network ***
  repeat
    messageID = GetNetworkMessage(netid)
  until messageID <> 0
  player = GetNetworkMessageInteger(messageID)
  square = GetNetworkMessageInteger(messageID)
  rem *** Update board array ***
  board[square] = player
  rem *** Delete the message ***
  DeleteNetworkMessage(messageID)
endfunction square

function UpdateScreen(square)
  rem *** Create the sprite to be placed on board ***
  spriteid = 20+square
  CreateSprite(spriteid,player)
  SetSpriteSize(spriteid,12,-1)
  rem *** Position sprite ***
  SetSpritePosition(spriteid,17+(spriteid-21) mod 3*27,21+(spriteid-21)/3*20)
  rem *** Change Next Player symbol ***
  SetSpriteFrame(9,3-player)
  Sync()
  rem *** Add 1 to the number of moves made ***
  inc movesmade
endfunction

function IsComplete()
  winner = 0
  line = 0
  for c = 1 to 2
    rem *** Check rows ***
    for row = 1 to 7 step 3
      if board[row]=c and board[row+1]=c and board[row+2]=c
        winner = c
        line = (row-1) / 3
      endif
    next row
  next c
  for c = 1 to 2
    rem *** Check columns ***
    for col = 1 to 3
      if board[col]=c and board[col+3]=c and board[col+6]=c
        winner = c
        line = 2+col
      endif
    next col
  next c
  rem *** Check diagonals ***
  if board[1]=c and board[5]=c and board[9]=c
    winner = c
    line = 6
  elseif board[3]=c and board[5]=c and board[7]=c
    winner = c
    line = 7
  endif
  result = winner*10+line
  if result = 0 and movesmade = 9
    result = 9
  endif
endfunction result

function FinishGame(status)
  rem *** Draw or win ***
  if status = 9
    GameDrawn()
  else
    ShowWinningLine(status)
  endif
endfunction

rem *****
rem *** Level 2 Functions ***
rem *****

function HandleButtons()
  rem *** If Host button pressed and not already joined ***
  if GetVirtualButtonPressed(1) = 1 and joined = 0
    rem *** Host the app ***
    netid = HostNetwork("TicTacToe", "Hostmachine",1026)
    rem *** Record as joined ***
    joined = 1
  endif
  rem *** If Client button pressed and not already joined ***
  if GetVirtualButtonPressed(2)=1 and joined = 0
    rem *** Join as client ***
    clientname$ = Str(GetSeconds())
    netid = JoinNetwork("TicTacToe",clientname$)
    rem *** Record as joined ***
    joined = 1
  endif
endfunction

function CloseGame()
  rem *** Display message ***
  CreateText(1,"Game needs two machines to play")
  SetTextSize(1,3)
  SetTextPosition(1,10,47)
  Sync()
  rem *** Wait 5 seconds then close game ***
  Sleep(5000)
endfunction

function GameDrawn()
  rem *** Load image used ***
  LoadImage(99,"TTTDraw.png")
  rem *** Display message ***
  CreateSprite(99,99)
  SetSpriteSize(99,60,-1)
  SetSpritePositionByOffset(99,50,47)
  Sync()
  rem *** Wait 5 seconds ***
  Sleep(5000)
endfunction

function ShowWinningLine(line)
  rem *** Load image used ***
  LoadImage(99,"TTTLine.png")
  rem *** Create sprite ***
  CreateSprite(99,99)
  SetSpriteSize(99,-1,60)

```

```

rem *** Check columns ***
for col = 1 to 3
  if board[col]=c and board[col+3]=c and board[col+6]=c
    winner = c
    line = 2+col
  endif
next col
rem *** Check diagonals ***
if board[1]=c and board[5]=c and board[9]=c
  winner = c
  line = 6
elseif board[3]=c and board[5]=c and board[7]=c
  winner = c
  line = 7
endif
next c
result = winner*10+line
if result = 0 and movesmade = 9
  result = 9
endif
endfunction result

function FinishGame(status)
  rem *** Draw or win ***
  if status = 9
    GameDrawn()
  else
    ShowWinningLine(status)
  endif
endfunction

rem *****
rem *** Level 2 Functions ***
rem *****

function HandleButtons()
  rem *** If Host button pressed and not already joined ***
  if GetVirtualButtonPressed(1) = 1 and joined = 0
    rem *** Host the app ***
    netid = HostNetwork("TicTacToe", "Hostmachine",1026)
    rem *** Record as joined ***
    joined = 1
  endif
  rem *** If Client button pressed and not already joined ***
  if GetVirtualButtonPressed(2)=1 and joined = 0
    rem *** Join as client ***
    clientname$ = Str(GetSeconds())
    netid = JoinNetwork("TicTacToe",clientname$)
    rem *** Record as joined ***
    joined = 1
  endif
endfunction

function CloseGame()
  rem *** Display message ***
  CreateText(1,"Game needs two machines to play")
  SetTextSize(1,3)
  SetTextPosition(1,10,47)
  Sync()
  rem *** Wait 5 seconds then close game ***
  Sleep(5000)
endfunction

function GameDrawn()
  rem *** Load image used ***
  LoadImage(99,"TTTDraw.png")
  rem *** Display message ***
  CreateSprite(99,99)
  SetSpriteSize(99,60,-1)
  SetSpritePositionByOffset(99,50,47)
  Sync()
  rem *** Wait 5 seconds ***
  Sleep(5000)
endfunction

function ShowWinningLine(line)
  rem *** Load image used ***
  LoadImage(99,"TTTLine.png")
  rem *** Create sprite ***
  CreateSprite(99,99)
  SetSpriteSize(99,-1,60)

```

```

rem *** Place horizontally ...***
if line <= 2
    SetSpriteAngle(99,90)
    SetSpritePositionByOffset(99,49,26+line*20)
endif
rem *** Place vertically ...***
if line >= 3 and line <= 5
    SetSpritePosition(99,20+(line-3)*27,16)
endif
rem *** Diagonal TL to BR ***
if line = 6
    SetSpriteAngle(99,135)
    SetSpritePositionByOffset(99,49,47)
endif
rem *** Diagonal BL to TR ***
if line = 7
    SetSpriteAngle(99,45)
    SetSpritePositionByOffset(99,49,47)
endif
Sync()
rem *** Wait for 5 seconds ***
Sleep(5000)
endfunction

```

Activity 21.20

No solution required.

Activity 21.21

No solution required.

Activity 21.22

The output is a string containing the HTML code contained in the file *index.html* downloaded from the **Digital Skills** website.

Activity 21.23

No solution required.

Activity 21.24

No solution required.

Activity 21.25

No solution required.

Bits and Pieces

In this Chapter:

- ☐ Date and Time Statements
- ☐ QR Coding
- ☐ Advert Insertion Statements
- ☐ Error Handling Statements
- ☐ Benchmarking Statements
- ☐ Resuming an App

Date and Time

Introduction

There are several commands which allow you to discover the date and time. These are covered in this chapter.

Standard Date Statements

GetCurrentDate()

The current date (as held within the executing device) can be found using the `GetCurrentDate()` statement which has the format shown in FIG-22.1.

FIG-22.1

string `GetCurrentDate()` ()

`GetCurrentDate()`

The string returned is in the form YYYY-MM-DD so, for example, the second of May 2012 would be returned as the string *2012-05-02*.

The program in FIG-22.2 gets the current date and creates three strings, one for each component of the date.

FIG-22.2

Extracting the
Elements of a Date

```
rem *** Get today's date ***
date$ = GetCurrentDate()

rem *** Extract the day, month and year ***
day$ = Right(date$,2)
year$ = Left(date$,4)
month$ = Mid(date$,6,2)

rem *** Display result in British format ***
do
  Print(day$+"/"+month$+"/"+year$)
  Sync()
loop
```

Activity 22.1

Start a new project called *Date01* and implement the code given in FIG-22.2.

If your country uses a different date format, modify the `Print()` statement in the program accordingly.

Test and save your project.

GetDayOfWeek()

The day on which the current date falls can be discovered using the `GetDayOfWeek()` statement (see FIG-22.3).

FIG-22.3

integer `GetDayOfWeek()` ()

`GetDayOfWeek()`

The date is returned as an integer value between 0 and 6. 0 represents Sunday, 1 Monday, etc.

The program in FIG-22.4 is a variation on *Date01*, this time adding the day of the week to the displayed date.

FIG-22.4

Displaying the Day of the Week

```
rem *** Day names ***
dim dayNames[6] as string =
  ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
  "Saturday"]

rem *** Get today's date ***
date$ = GetCurrentDate()

rem *** Extract the day, month and year ***
day$ = Right(date$, 2)
year$ = Left(date$, 4)
month$ = Mid(date$, 6, 2)

rem *** Display result in British format ***
do
  Print(dayNames[GetDayOfWeek()]+ " "+day$+" / "+month$+" / "+year$)
  Sync()
loop
```

Activity 22.2

Modify *Date01* to match the code in FIG-22.4.

Test and save your project.

GetLeapYear()

The `GetLeapYear()` function returns 1 if the current year is a leap year (a 366 day year), otherwise zero is returned. The statement's format is shown in FIG-22.5.

FIG-22.5

`GetLeapYear()`

integer `GetLeapYear` ()

For example, the line

```
result = GetLeapYear()
```

would set result to 1 if run in the years 2012, 2016, 2020, etc and return 0 for the years 2013, 2014, 2015, 2017, etc.

Unix Date Statements

The Unix operating system dates everything in seconds from midnight 1st January 1970 using the Coordinated Universal Time (UTC) standard. It may not be exactly accurate with actual time since it does not take into account leap seconds which are occasionally added.

AGK offers statements which convert a Unix time to day, month and year.

GetDayFromUnix()

To find the day element of a date based on Unix time, use the `GetDayFromUnix()` statement (see FIG-22.6).

FIG-22.6

`GetDayFromUnix()`

integer `GetDayFromUnix` (`iunixtime`)

where

`iunixtime` is an integer value giving the seconds since 1/1/1970.

GetMonthFromUnix()

To find the month element of a date based on Unix time, use the `GetMonthFromUnix()` statement (see FIG-22.7).

FIG-22.7

`GetMonthFromUnix()`

integer `GetMonthFromUnix` (`iunixtime`)

where

`iunixtime` is an integer value giving the seconds since 1/1/1970.

GetYearFromUnix()

To find the year element of a date based on Unix time, use the `GetYearFromUnix()` statement (see FIG-22.8).

FIG-22.8

`GetYearFromUnix()`

integer `GetYearFromUnix` (`iunixtime`)

where

`iunixtime` is an integer value giving the seconds since 1/1/1970.

The program in FIG-22.9 makes use of these routines to convert the Unix time 1335893487 to a standard date.

FIG-22.9

Obtaining a Standard
Date from Unix Time

```
rem *** Unix Date ***

rem *** Convert to day, month, year ***
day = GetDayFromUnix(1335893487)
month = GetMonthFromUnix(1335893487)
year = GetYearFromUnix(1335893487)

rem *** Display result in British format ***
do
    Print(Str(day) + "/" + Str(month) + "/" + Str(year))
    Sync()
loop
```

Activity 22.3

Start a new project called *Date02* and implement the code in FIG-22.9.

Test and save your project.

Time Statements

If you need the time in hours, minutes and seconds, then there are a set of routines for that too.

GetCurrentTime()

The current time can be obtained as a string in the format HH-MM-SS using the `GetCurrentTime()` statement (see FIG-22.10).

FIG-22.10

string `GetCurrentTime` ()

`GetCurrentTime()`

Use the `GetStringToken()` function to extract the parts of the time from the string.

Activity 22.4

Start a new project called *Time01* which extracts the hours, minutes and seconds as integer values from the string returned by `GetCurrentTime()`.

Calculate and display the number of seconds passed since the start of the day.

Test and save your project.

GetHoursFromUnix()

The hours component of a time can be derived from a Unix time using `GetHoursFromUnix()` (see FIG-22.11).

FIG-22.11

integer `GetHoursFromUnix` () `iunixtime` ()

`GetHoursFromUnix()`

where

`iunixtime` is an integer value giving the seconds since 1/1/1970.

GetMinutesFromUnix()

The minutes component of a time can be derived from a Unix time using `GetMinutesFromUnix()` (see FIG-22.12).

FIG-22.12

integer `GetMinutesFromUnix` () `iunixtime` ()

`GetMinutesFromUnix()`

where

`iunixtime` is an integer value giving the seconds since 1/1/1970.

GetSecondsFromUnix()

The seconds component of a time can be derived from a Unix time using `GetSecondsFromUnix()` (see FIG-22.13).

FIG-22.13

integer `GetSecondsFromUnix` () `iunixtime` ()

`GetSecondsFromUnix()`

where

`iunixtime` is an integer value giving the seconds since 1/1/1970.

Activity 22.5

Start a new project called *Time02* and, using the same Unix time value shown in FIG-22.9, display the time of day in hours, minutes and seconds.

Test and save your project.

GetUnixFromDate()

To discover the Unix time for a given date and time, we can use `GetUnixFromDate()` (see FIG-22.14).

FIG-22.14 `GetUnixFromDate()`

integer `GetUnixFromDate` ((`iyear` , `imonth` , `iday` , `ihour` , `imin` , `isec`)

where

iyear, **imonth**, **iday** are integer values giving the date.

ihour, **imin**, **isec** are integer values giving the time.

A typical call to the statement might be:

```
seconds = GetUnixFromDate(2012,5,2,11,48,20)
```

Dates before 1/1/1970 will return negative values. On 32 bit systems, the dates must lie between 1901 and 2038.

Summary

- Use `GetCurrentDate()` to obtain today's date as a string in the format YYYY-MM-DD.
- Use `GetDayOfWeek()` to obtain the day of the week in numeric form (0: Sunday, 1: Monday, etc.)
- Use `GetLeapYear()` to check if the current year is a leap year.
- Unix time is measured in seconds from midnight on 1/1/1970.
- Use `GetDayFromUnix()`, `GetMonthFromUnix()`, and `GetYearFromUnix()` to get the day, month and year of a Unix time.
- Use `GetCurrentTime()` to obtain the current time of day as a string in the format HH-MM-SS.
- Use `GetHoursFromUnix()`, `GetMinutesFromUnix()`, and `GetSecondsFromUnix()` to get the hours, minutes and seconds components of a Unix time.
- Use `GetUnixFromDate()` to convert a standard date and time into Unix time.

Introduction

Quick Response code (or QR code, as it is more usually called), is a two dimensional barcoding system for coding text, binary or Japanese Kanji characters. A typical QR code is shown in FIG-22.15.

FIG-22.15

A QR Barcode



The coding system was first used in the Japanese car industry but has since become widely popular in many other areas. The reason for its growing popularity is that it can be decoded using a smartphone or tablet with the appropriate software.

AGK contains two QR code-related statements. One to create a QR code and the other to decode a QR code.

QR Code Statements

EncodeQRCode()

To create a QR code for a piece of text, use `EncodeQRCode()` (see FIG-22.16).

FIG-22.16

EncodeQRCode()

integer `EncodeQRCode` (`text` , `ierr`)

where

text is a string giving the text to be encoded.

ierr is an integer value (0 to 3) giving the level of automatic error correction to be embedded in the code (0: lowest, 3: highest).

The function creates an image of the QR code and returns the ID assigned to that image. To see the QR code, all you need to do is assign the image to a sprite.

The program in FIG-22.17 allows the user to enter text and that text is then displayed in the form of a QR code.

FIG-22.17

Creating a QR Code

```
rem *** Convert Text to QR Code ***  
  
rem *** Create focused edit box ***  
CreateEditBox(1)  
SetEditBoxSize(1,40,5)  
SetEditBoxPosition(1,20,10)  
SetEditBoxFocus(1,1)  
SetEditBoxTextSize(1,2)
```



FIG-22.17

(continued)

Creating a QR Code

```

rem *** Create empty sprite ***
CreateSprite(1,0)
SetSpriteSize(1,40,-1)
SetSpritePosition(1,30,30)

do
    rem *** If edit box loses focus ***
    if GetEditBoxChanged(1) = 1
        rem *** Get its text ... ***
        text$ = GetEditBoxText(1)
        rem *** ... and convert to QR ***
        id = EncodeQRCode(text$,1)
        SetSpriteImage(1,id)
    endif
    Sync()
loop

```

Activity 22.6

Start a new project called *QRCode01* and implement the code given in FIG-22.17.

Test and save your project.

DecodeQRCode()

To decode an existing QR code, use `DecodeQRCode()` (see FIG-22.18).

FIG-22.18

DecodeQRCode()

string `DecodeQRCode` (`imgId`)

where

imgId is an integer value giving the ID of the image to be decoded.

The function returns an empty string if no QR code was found in the image or the code could not be interpreted.

The program in FIG-22.19 uses a device's built-in camera to scan a QR image. If the image contains a web address, the browser is opened at that site, otherwise the QR text is displayed. If you don't have a camera on your device, a default QR code image is loaded.

FIG-22.19

Reading a QR Code

```

rem *** Get QR from Camera and load Site ***

rem *** Get an image from the camera ***
id = GetCameraImage()
rem *** If no image, use default image ***
if id = 0
    id = LoadImage("QRExample.png")
endif
rem *** Show image in sprite ***
CreateSprite(1,id)
SetSpriteSize(1,20,-1)
SetSpritePosition(1,20,20)

```



FIG-22.19

(continued)

Reading a QR Code

```

rem *** If text starts "www." ***
rem *** open browser at site ***
if Lower(Left(text$,4)) = "www."
    OpenBrowser(text$)
else
    rem *** else show text from QR ***
    CreateText(1,text$)
endif
do
    Sync()
loop

function GetCameraImage()
    rem *** If camera software operational ***
    if ShowImageCaptureScreen() = 1
        rem *** Wait until an image has been captured ***
        repeat
            Sync()
        until IsCapturingImage() = 0
        rem *** Get ID assigned to captured image ***
        id = GetCapturedImage()
    else
        id = 0
    endif
endfunction id

```

Activity 22.7

Start a new project called *QRCode02* and implement the code given in FIG-22.19.

If your device does not have a built-in camera, copy the file *AGKDownloads/Chapter22/QRExample.png* to the *media* folder.

When running the program, take a snap of the QR code printed at the start of this section. This will be decoded to give you access to a web address.

Save your project.

Summary

- A QR code is a two-dimensional barcoding system which can store text, binary or Japanese Kanji characters.
- Use `EncodeQRCode()` to create a QR code image from specified text.
- Use `DecodeQRCode()` to convert a QR code image to a string.

Introduction

If you intend to give your apps away for free but would still like to earn money, then the way to go is to add advertisements to your product.

The first stage in adding adverts to your app is to sign up with an advertising agency that specialises in this area. This will supply you with a unique code which must be used when setting up the advertising areas within your app. The current version of AGK (May 2012) specifically contains a command for using the **inneractive** agency.

The actual advert that is inserted will be up to the advertising agency and they will pay you for each click on the advert.

AGK contains several commands to allow you to create, position and delete advertising space within your app.

Ad Statements

SetInneractiveDetails()

The `SetInneractiveDetails()` function sets up the necessary details within your app to allow live adverts to be placed within the area defined. To use this statement you will need to have created an account with **inneractive** (you can find them at inner-active.com).

The `SetInneractiveDetails()` statement has the format shown in FIG-22.20.

FIG-22.20

SetInneractiveDetails



where


icode is an integer value giving the account code supplied to you when you signed up with **inneractive**.

CreateAdvert()

To create an advertising window within your app, use the `CreateAdvert()` statement (see FIG-22.21).

FIG-22.21

CreateAdvert()



where

itype is an integer value (0 only - at the moment) giving the type of advert space being created (0: 320x50 pixel).

ihorz is an integer value (0,1 or 2) giving the horizontal positioning of the advert area (0: left, 1: centre, 2: right).

ivert is an integer value (0,1 or 2) giving the vertical positioning of the advert area (0: top, 1: centre, 2: bottom).

itest is an integer value (0 or 1) used to specify if the displayed advert is for test purposes only (1) or is a real advert (0).

IMPORTANT: Make sure you always use the test advert option during the development of your app, only changing to the real advert option when you are ready to deploy your application through app stores. If the advertiser suspects you are clicking a real advert during development, they will close your account and you will not be able to benefit from advertising revenue with that agency.

SetAdvertPosition()

Some advertisers allow greater control about positioning and sizing of the advert. Where this is possible, you can use the `SetAdvertPosition()` statement (see FIG-22.22).

FIG-22.22

`SetAdvertPosition (x , y , fwidth)`

`SetAdvertPosition()`

where

x,y are real values giving the coordinates for the top-left corner of the advert.

fwidth is a real number giving the width of the advert. The height of the advert will be adjusted automatically to maintain the width-to-height ratio.

DeleteAdvert()

If you want to remove an advertising area from your app, use the `DeleteAdvert()` statement (see FIG-22.23).

FIG-22.23

`DeleteAdvert ()`

`DeleteAdvert()`

Obviously, it is not possible to give you a working example of an advert in place within an app, but the general approach is encapsulated in the function shown in FIG-22.24.

FIG-22.24

General approach to placing an advert in your App

```
function PlaceAdvert()  
    SetInnerActiveDetails("agency code goes here")  
    CreateAdvert(0,1,2,1) //Test advert - change later  
    SetAdvertPosition(0,89,100) //Position as required  
endfunction
```

Summary

- To add adverts to your app, you must first sign up with an advertising agency such as **inneractive**.
- Use `SetInnerActiveDetails()` to add your **inneractive**-assigned code to your program. This enables test and live adverts.
- Use `CreateAdvert()` to create advertising space in your app.

- Make sure you use test mode for your adverts when developing and testing your app.
- Change the advert option to live when you are about to submit your app to the app store.
- Use `SetAdvertPosition()` to position and resize adverts where this is allowed.
- Use `DeleteAdvert()` to remove the advert area from your app.

Introduction

Somewhere in all the projects you've been working on you probably had your program stop and display an error message. Perhaps a line such as *Sprite 4 does not exist at line 6*.

You can gain some control over how your program handles this type of situation using the various error control commands.

Error Handling Statements

SetErrorMode()

To specify how you want your program to react to these run-time errors, use the `SetErrorMode()` statement (see FIG-22.25).

FIG-22.25

SetErrorMode()

`SetErrorMode (imode)`

where

imode is an integer value (0,1 or 2).

A value of 0 will cause the program to ignore the error entirely and attempt to carry on executing your code. You can use the other error-handling statements (see below) to discover what error occurred.

Setting *imode* to 1 will cause the program to report the error in a message window, but the program will attempt to continue.

Throwing an exception is a method of handling error situations in C++ and similar languages. It cannot be handled in your BASIC program.

A value of 2 causes the program to terminate, reporting the error and throwing an exception (which is also displayed in a message box).

GetErrorOccurred()

Once you have set the error mode, you can detect that an error has occurred using `GetErrorOccurred()` (see FIG-22.26).

FIG-22.26

GetErrorOccurred()

integer `GetErrorOccurred ()`

The function returns 1 if an error has occurred, otherwise zero is returned.

GetLastError()

A string giving a description of the last error to have occurred can be retrieved using `GetLastError()` (see FIG-22.27).

FIG-22.27

GetLastError()

string `GetLastError ()`

A simple program making use of these three statements is shown in FIG-22.28.

FIG-22.28

Using Error-Handling
Statements

```
rem *** Handling errors ***

rem *** Ignore errors ***
SetErrorMode(0)

rem *** Try to load a non-existent image ***
LoadImage(1,"MyPic.png")

rem *** Report any error ***
if GetErrorOccurred() = 1
    Print(GetLastError())
    Sync()
    Sleep(2000)
endif

do
    Print ("Program continues")
    Sync()
loop
```

Activity 22.8

Start a new project called *ErrorHandling* and implement the code given in FIG-22.28.

Test the program as it attempts to load the unavailable file.

Re-test the program using error modes 1 and 2, observing the difference between each option.

Summary

- Use `SetErrorMode()` to control how a program responds to a runtime error.
- Use `GetErrorOccurred()` to detect if an error has occurred during the execution of the program.
- Use `GetLastError()` to access a message describing the last error to have occurred.

Benchmarking

Introduction

Any game that involves real-time movement requires a reasonably high frame rate. Without that, movement will look unnatural and jerky.

If your game has a poor frame rate, AGK contains various benchmarking statements that allow you to discover timings and counts for various elements in your program. These timings can help you identify what aspects of the program are causing poor performance and where to make changes to your code or game design. The counts may be useful for finding out what components in your program are causing any unacceptable timings.

Benchmarking Statements

GetDrawingSetupTime()

You can find out the time taken (in seconds) to set up the required visual elements of a screen using `GetDrawingSetupTime()` (see FIG-22.29).

FIG-22.29

float `GetDrawingSetupTime()`

`GetDrawingSetupTime()`

The time returned includes the time taken to send details to the graphics processing unit (GPU).

GetDrawingTime()

To discover the time taken (in seconds) to swap screen buffers (as the back buffer becomes the front buffer), use `GetDrawingTime()` (see FIG-22.30).

FIG-22.30

float `GetDrawingTime()`

`GetDrawingTime()`

The time will include any delays caused by waiting for the GPU to become ready.

GetPhysicsTime()

If you are using the physics engine, you can find out the time spent calculating the physics for each frame using `GetPhysicsTime()` (see FIG-22.31).

FIG-22.31

integer `GetPhysicsTime()`

`GetPhysicsTime()`

GetUpdateTime()

The time taken to do everything other than physics in the last frame build-up is returned by the `GetUpdateTime()` function (see FIG-22.32).

FIG-22.32

integer `GetUpdateTime()`

`GetUpdateTime()`

GetManagedSpriteCount()

AGK contains a sprite manager. The sprite manager is responsible for handling all

the sprites currently in existence. Amongst other things, it ensures that animated sprites are displaying the correct frame and identifies which sprites are on-screen and hence need to be drawn.

The number of sprites being handled by the program can be discovered using the `GetManagedSpritesCount()` statement (see FIG-22.33).

FIG-22.33

`GetManagedSpriteCount()` integer `GetManagedSpriteCount()` ()

GetManagedSpriteDrawnCount()

If a program contains sprites that are currently off-screen or invisible, then there will be a difference between the number of sprites being managed and the number being drawn. To discover how many sprites need to be drawn for the current frame, use `GetManagedSpriteDrawnCount()` (see FIG-22.34).

FIG-22.34

`GetManagedSpriteDrawnCount()` integer `GetManagedSpriteDrawnCount()` ()

GetManagedSpriteDrawCalls()

To find the number of OpenGL calls required to create the visible sprites, use the `GetManagedSpriteDrawCalls()` statement (see FIG-22.35).

FIG-22.35

`GetManagedSpriteDrawCalls()` integer `GetManagedSpriteDrawCalls()` ()

OpenGL (Open Graphics Library) is a set of multi-platform graphics routines and is used by AGK to draw all visual components.

GetManagedSpriteSortedCount()

A count of the changes to the sprites' depth or texture can be obtained using `GetManagedSpriteSortedCount()` (see FIG-22.36).

FIG-22.36

`GetManagedSpriteSortedCount()` integer `GetManagedSpriteSortedCount()` ()

GetParticleDrawnPointCount()

If your program is using particles (see Chapter 13), then the number of particles drawn using the point method in the last frame can be found using `GetParticleDrawnPointCount()` (see FIG-22.37).

FIG-22.37

`GetParticleDrawnPointCount()` integer `GetParticleDrawnPointCount()` ()

GetParticleDrawnQuadCount()

When particles become larger (more than about 64 screen pixels), then they are drawn using the quad method (a slower technique than point drawing). The number of particles drawn using the quad method can be determined using `GetParticleDrawnQuadCount()` (see FIG-22.38).

FIG-22.38

`GetParticleDrawnQuadCount()` integer `GetParticleDrawnQuadCount()` ()

GetPixelsDrawn()

The approximate number of pixels drawn in the last frame can be discovered using `GetPixelsDrawn()` (see FIG-22.39).

FIG-22.39

integer `GetPixelsDrawn` ()

`GetPixelsDrawn()`

The program in FIG-22.40 is the apples and oranges program we last saw in Chapter 20 with additional statements to display the various benchmark timings and counts values.

FIG-22.40

Displaying Benchmark
Values

```
rem *** Benchmark Counts and Times ***

rem *** Set background colour ***
SetClearColor(150,150,170)
Sync()

rem *** Load images ***
LoadImage(1,"Apple.png")
LoadImage(2,"Orange.png")
LoadImage(3,"GreenApple.png")

rem *** Create 10 apple sprites ***
CreateSprite(1,1)
SetSpriteSize(1,6,-1)
SetSpritePosition(1,Random(0,92),Random(0,92))
rem *** Red apples ***
for c = 2 to 10
    CloneSprite(c,1)
    SetSpritePosition(c,Random(0,92),Random(0,92))
next c
rem *** Change 5 apples to green ***
for c = 6 to 10
    SetSpriteImage(c,3)
next c

rem *** Create 10 orange sprites ***
CreateSprite(11,2)
SetSpriteSize(11,6,-1)
SetSpritePosition(11,Random(0,92),Random(0,92))
for c = 12 to 20
    CloneSprite(c,11)
    SetSpritePosition(c,Random(0,92),Random(0,92))
next c

rem *** Group apples ***
for c = 1 to 10
    SetSpriteGroup(c,-1)
next c
rem *** Set the category for red apples ***
for c = 1 to 5
    SetSpriteCategoryBits(c,%100)
next c

rem *** Set the category for green apples ***
for c = 6 to 10
    SetSpriteCategoryBits(c,%1000)
next c
```



FIG-22.40

(continued)

Displaying Benchmark
Values

```

rem *** Group and categorise oranges ***
for c = 11 to 20
    SetSpriteGroup(c,-2)
rem *** Set the category for oranges ***
SetSpriteCategoryBits(c,%10)
rem *** Set the categories with which oranges can collide ***
SetSpriteCollideBits(c,%101)
next c

rem *** Switch on physics ***
for c = 1 to 20
    SetSpriteShape(c,1)
    SetSpritePhysicsOn(c,2)
    angle = Random(0,359)
    SetSpritePhysicsVelocity(c,Cos(angle)*50,Sin(angle)*50)
    SetSpritePhysicsRestitution(c,1)
next c

rem *** No gravity ***
SetPhysicsGravity(0,0)

rem *** Record start time ***
time = GetSeconds()
do
    BenchMarkTimes()
    BenchMarkCounts()
rem *** If 10 seconds passed ***
if GetSeconds() - time = 10
    rem *** Set all oranges to collide with green apples ***
    for c = 11 to 20
        SetSpriteCollideBit(c,4,1)
    next c
endif
    Sync()
loop

function BenchMarkTimes()
    Print("Drawing setup time : "+Str(GetDrawingSetupTime()))
    Print("Drawing time       : "+Str(GetDrawingTime()))
    Print("Physics time       : "+Str(GetPhysicsTime()))
    Print("Update time        : "+Str(GetUpdateTime()))
endfunction

function BenchMarkCounts()
    Print("Managed sprite count      : ")
    Print("↳+Str(GetManagedSpriteCount()))
    Print("Managed sprites drawn count : ")
    Print("↳+Str(GetManagedSpriteDrawnCount()))
    Print("Managed sprites draw calls : ")
    Print("↳+Str(GetManagedSpriteDrawnCalls()))
    Print("Managed sprite sorted count : ")
    Print("↳+Str(GetManagedSpriteSortedCount()))
    Print("Particles drawn (points) count: ")
    Print("↳+Str(GetParticleDrawnPointCount()))
    Print("Particles drawn (quad) count : ")
    Print("↳+Str(GetParticleDrawnQuadCount()))
    Print("Pixels drawn count          : "+Str(GetPixelsDrawn()))
endfunction

```

Activity 22.9

Update your *PhysicsGroup* project (which you created in Activity 20.27) to include the Benchmark functions given in FIG-22.40.

Add calls to the functions within the program's `do..loop` and observe the values displayed.

Save your project.

Summary

- Use `GetDrawingSetupTime()` to discover the time required to set up the required visual elements of a screen.
- Use `GetDrawingTime()` to discover the time taken to swap screen buffers.
- Use `GetPhysicsTime()` to find out the time spent calculating the physics for each frame.
- Use `GetUpdateTime()` to find the time taken to do everything other than physics in the last frame build-up.
- Use `GetManagedSpriteCount()` to discover the number of sprites being handled by the sprite manager during each frame.
- Use `GetManagedSpriteDrawnCount()` to discover the number of sprites being drawn during each frame.
- Use `GetManagedSpriteDrawCalls()` to find the number of calls to OpenGL routines during each frame.
- Use `GetManagedSpriteSortedCount()` to find the number of changes to sprite textures or layers during each frame.
- Use `GetParticleDrawnPointCount()` to find the number of particles drawn using the point method during each frame.
- Use `GetParticleDrawnQuadCount()` to find the number of particles drawn using the quad method during each frame.
- Use `GetPixelsDrawn()` to find the number of pixels drawn during each frame.

Paused Apps

If your app is being run on a smartphone, it will be halted by any incoming phone call. Your app will then be passed to the background while you deal with the call. When the call is over, the app will be restarted. Even when using a tablet with only a Wi-Fi connection, the user may jump to another app in the middle of running your game. On a PC, your app will continue to run even if another window gains focus.

When your app regains control of the device, it is best to display a pause screen to allow the user to decide exactly when the game is to restart.

GetResumed()

To detect that your app has just regained control after being held in the background, use `GetResumed()` (see FIG-22.41).

FIG-22.41

`GetResumed()`

integer `GetResumed` ()

The function returns 1 if the app has just been reactivated, otherwise zero is returned.

The program in FIG-22.42 is a modification of the bat and ball game you last used in Activity 17.33. In this version, a paused screen will appear if the app restarts after being halted.

FIG-22.42

Resuming an App

```
rem *** Bat and Ball with Pause ***

rem *** Load images ***
LoadImage(1,"Ball.png")
LoadImage(2,"Bat.png")

rem *** Create ball Sprite ***
CreateSprite(1,1)
SetSpriteSize(1,4,-1)
SetSpritePosition(1,48,5)

rem ** Create bat sprite ***
CreateSprite(2,2)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,47.5,95)

rem *** Record bat's x-coordinate ***
batx = 47.5

rem *** Create virtual buttons ***
AddVirtualButton(1,5,94,10)
AddVirtualButton(2,95,94,10)
SetVirtualButtonVisible(1,0)
SetVirtualButtonVisible(2,0)
Sync()

rem *** Wait 2 seconds before starting ***
Sleep(2000)

rem *** Set ball's velocity***
yoffset# = Random(3,8)/10.0
xoffset# = (Random(0,40)-20)/10.0
```



FIG-22.42

(continued)

Resuming an App

```

rem *** Set game state to playing (1) ***
gamestate = 1

rem *** Play until ball leaves bottom of screen ***
repeat
    rem *** If resumed, show pause message ***
    if GetResume() = 1
        ShowPausedScreen()
    endif
    rem *** Redraw ball's position ***
    SetSpritePosition(1,GetSpriteX(1)+xoffset#,GetSpriteY(1)
    ↵+yoffset#)
    rem *** If the sprite hits the left or right sides change
    ↵xoffset ***
    if GetSpriteX(1)<=0 or GetSpriteX(1)>= 100- GetSpriteWidth(1)
        xoffset# = -xoffset#
    rem *** If ball hits top or bat, change yoffset ***
    elseif GetSpriteY(1)<=0 or GetSpriteCollision(1,2) = 1
        yoffset# = -yoffset#
    rem *** If sprite passes bottom edge, end game ***
    elseif GetSpriteY(1) > 100
        DeleteSprite(1)
        gamestate = 0
    endif
    rem *** Move bat ***
    if GetVirtualButtonState(1) = 1
        dec batx
    elseif GetVirtualButtonState(2) = 1
        inc batx
    endif
    SetSpritePosition(2,batx,GetSpriteY(2))
    Sync()
until gamestate = 0
end

function ShowPausedScreen()
    rem *** Show paused message ***
    LoadImage(99,"PressToContinue.png")
    CreateSprite(99,99)
    SetSpriteSize(99,60,-1)
    SetSpritePositionByOffset(99,50,50)
    rem *** Not ready to restart app ***
    restart = 0
    rem *** Wait until message pressed ***
    repeat
        if GetPointerPressed() = 1
            if GetSpriteHit(GetPointerx(),GetPointerY())=99
                DeleteSprite(99)
                DeleteImage(99)
                restart = 1
            endif
        endif
        Sync()
    until restart = 1
endfunction

```

Activity 22.10

Reload your *BatandBall* project and modify it by adding the pause option.

Copy the file *PressToContinue.png* to the project's *media* folder.

Run the program on your smart device and select a different app when the game starts.

Return to the game and check that the pause message has appeared.

Select the message to resume the game.

Save your project.

Solutions

Activity 22.9

No solution required.

Activity 22.1

No solution required.

Activity 22.2

No solution required.

Activity 22.3

No solution required.

Activity 22.4

Code for *Time01*:

```
rem *** Current Time to Seconds ***

do
  rem ** get time as a string **
  time$ = GetCurrentTime()
  rem *** How many tokens in string(should be 3)
  ↵***
  tokens = CountStringTokens(time$,":")
  rem *** Multiply number by seconds in time unit
  ↵***
  rem *** Start with seconds in an hour ***
  multiplier = 60*60
  rem *** Start total at zero ***
  totalsecs = 0
  rem *** For each token ***
  for c = 1 to tokens
    rem *** Convert it to a number and multiply
    ↵***
    rem *** by seconds in that time unit
    ↵***
    t = Val(GetStringToken(time$,":",c))*
    ↵multiplier
    rem *** Add to total seconds ***
    totalsecs = totalsecs + t
    rem *** Next time unit has 60 times less
    ↵seconds ***
    multiplier = multiplier/60
  next c
  rem *** Display result ***
  Print(time$+" is " + Str(totalsecs)+" seconds")
  Sync()
loop
```

Activity 22.5

Code for *Time02*:

```
rem *** Unix Time ***

hours = GetHoursFromUnix(1335893487)
mins = GetMinutesFromUnix(1335893487)
secs = GetSecondsFromUnix(1335893487)
do
  Print(Str(hours)+" "+Str(mins)+" "+Str(secs))
  Sync()
loop
```

Activity 22.6

No solution required.

Activity 22.7

No solution required.

Activity 22.8

No solution required.

Activity 22.10

No solution required.

3D Graphics

In this Chapter:

- ☐ Defining 3D Space
- ☐ Elements of a 3D Model
- ☐ Model Resolution
- ☐ 3D Primitives
- ☐ Texturing Models
- ☐ Local Axes
- ☐ Lighting
- ☐ Camera Control
- ☐ Shaders

Concepts and Terminology

Introduction

What is 3D?

With the increased popularity of so-called 3D movies, some people may be confused by the term 3D when used in the context of computer models. A 3D movie should more correctly be called a stereoscopic movie. Although it gives some impression of the 3D world we live in with objects appearing to come out of the screen or recede into it, the illusion is limited. For example, you can't change your viewing position in such a way as to see behind objects. On the other hand, 3D computer models, although they lack the stereoscopic effect of the movies (unless you have a 3D screen and the appropriate software), permit you to change your viewpoint thereby allowing you to see behind, above, or below objects on the screen.

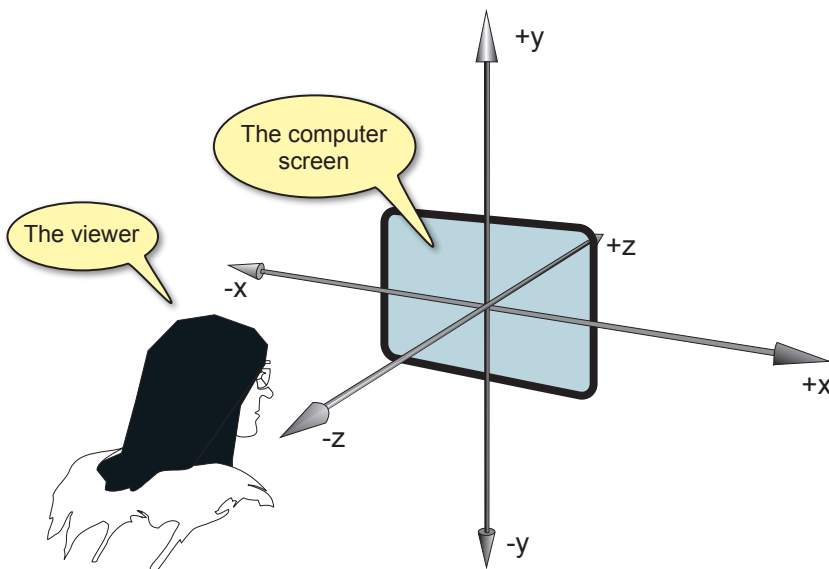
3D Space Axes

So far, all our games have been strictly two-dimensional to reflect the two-dimensional nature of a screen but AGK also allows us to simulate the third dimension - depth.

In 2D space we have the x and y axes which allow us to specify the position of any point on a surface. For three-dimensional space we need to add a third axis: the **z-axis**. When looking directly at a screen, the x-axis spans the width of the screen, the y-axis the height of the screen, and the z-axis can be imagined as a line coming straight out of the screen towards the viewer adding depth (see FIG-23.1).

FIG-23.1

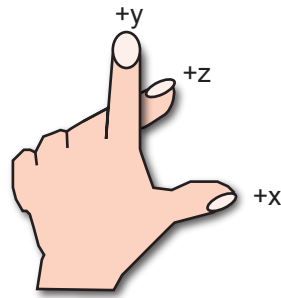
3D Axes Orientation



Notice that it is the negative end of the z-axis that protrudes from the screen. This setup is known as a **left-handed coordinate system** because, using your thumb and first two fingers of your left-hand, you can point in the positive direction of all three axes (see FIG-23.2).

FIG-23.2

The Left-Handed
Coordinate System



This arrangement of the axes used by AGK is by no means universally agreed upon, so you may find some modelling packages use different arrangements. However, if you are using Milkshape to create your models, it makes use of the same axes layout as AGK.

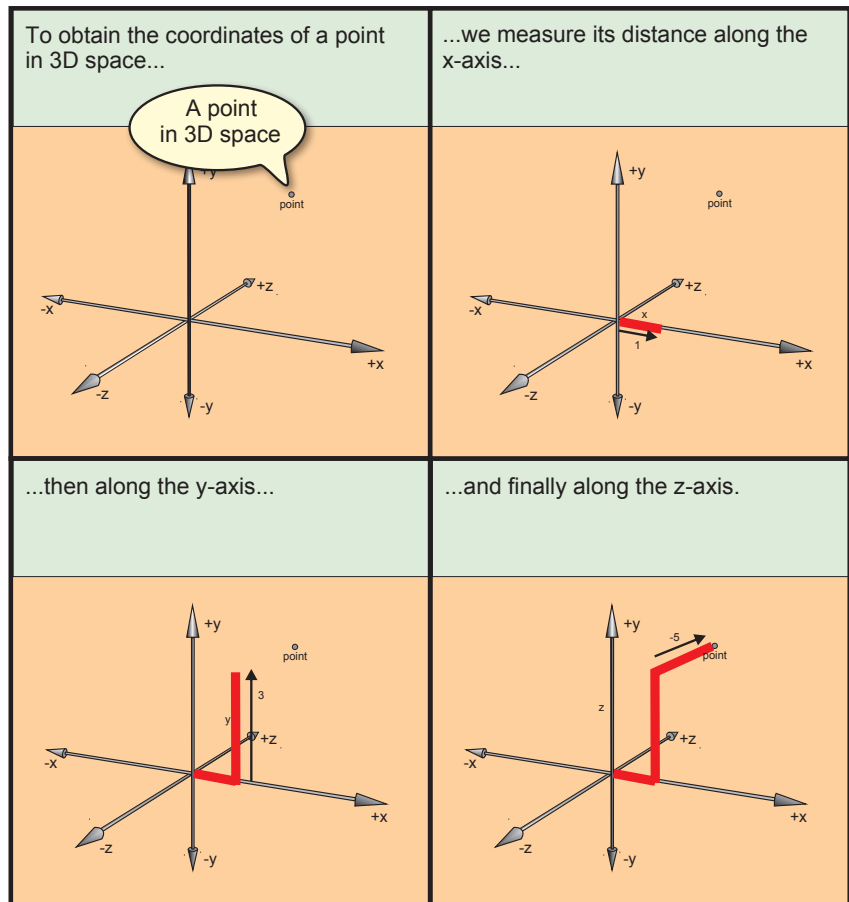
Another important point to note is that, when working in 3D space, the origin is assumed to be at the centre of the screen and not in the top-left corner as in 2D apps.

Defining a Point in 3D Space

A point in 3D space is defined by its distance along each of the three axes. For example, the point (1,3,-5) is at position 1 on the x-axis, position 3 on the y-axis and position -5 on the z-axis (see FIG-23.3).

FIG-23.3

Defining a Point in 3D
Space

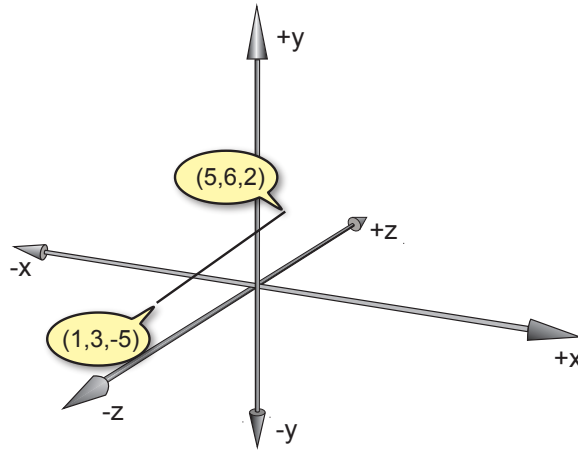


Lines

A line in 3D space, like one in 2D space, is defined by its start and end points. Of course, this time, those points are themselves defined using three coordinates, rather than two. For example, the line defined by the points (1,3,-5), (5,6,2) is shown in FIG-23.4.

FIG-23.4

Defining a Line in 3D Space



The displacement of a line along a given axis is the value of the difference between the end and start points for that particular axis. In the line shown above, the displacements are:

$$\begin{aligned} \text{x displacement} &= x_{\text{end}} - x_{\text{start}} \\ &= 5 - 1 \\ &= 4 \\ \text{y displacement} &= y_{\text{end}} - y_{\text{start}} \\ &= 6 - 3 \\ &= 3 \\ \text{z displacement} &= z_{\text{end}} - z_{\text{start}} \\ &= 2 - (-5) \\ &= 7 \end{aligned}$$

In 2D space, the length of a line is calculated as the square root of the $x_{\text{disp}}^2 + y_{\text{disp}}^2$. In 3D, the length of the line is calculated as

$$\text{length} = \sqrt{x_{\text{disp}}^2 + y_{\text{disp}}^2 + z_{\text{disp}}^2}$$

Activity 23.1

Calculate the length of the line defined in FIG-23.4.

AGK uses a line when defining ray traces.

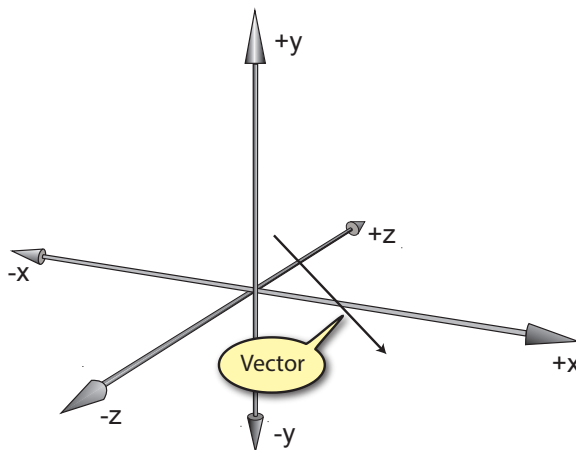
Vectors

At first glance, it is easy to confuse a vector with a line, but they are quite different creatures. In computer graphics, a vector is used to give information about direction and magnitude - this last characteristic is typically used to define speed or force.

Graphically a vector is shown as an arrowed line with the arrow giving the direction of travel (see FIG.23.5).

FIG-23.5

A Vector in 3D Space



Another difference between lines and vectors is that a vector has no defined position in space - it defines only direction and magnitude.

Typically, details of a 3D vector are given as three numbers representing its displacements in the three dimensions.

The magnitude is calculated using the same formula as we used earlier for the length of a line.

Some mathematical calculations require a vector whose magnitude is exactly 1. A vector with a magnitude of 1 is known as a **unit vector**. Of course, changing a vector to have a magnitude of exactly 1, although it will effect the values of the offsets, has no effect on the direction of the vector.

When a vector is used to indicate the velocity (speed and direction) of a moving object, the magnitude also represents a given unit of time such as one second or the time taken to produce a single screen frame. For example, if we wanted to use a vector to represent the velocity of a rocket taking off vertically at a speed of 10 units per frame, then the vector's offsets would be $[0, 10, 0]$. Notice that in mathematics, the values in a vector are written enclosed in square brackets

In other cases, such as when a vector is used to indicate the direction of the light falling on a 3D model, the magnitude of the vector is unimportant.

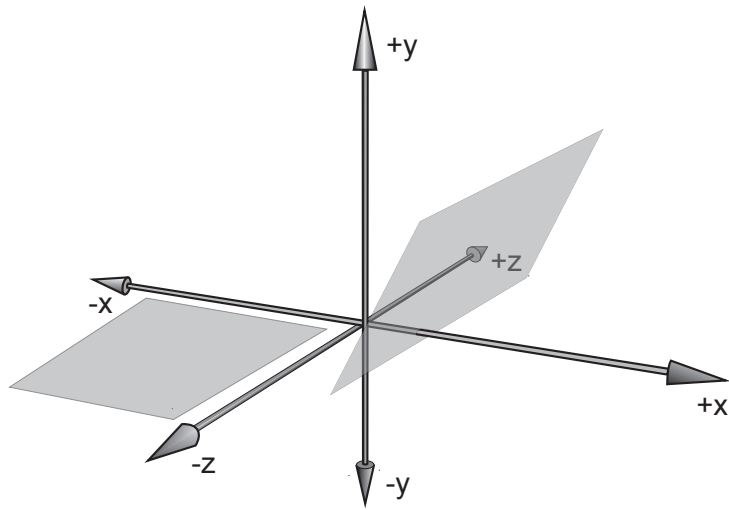
Planes

A plane is a flat two-dimensional surface within 3D space. You can think of it as something a bit like a sheet of paper.

A plane can lie at any angle to the axes. The two shaded areas in FIG-23.6 represent two planes.

FIG-23.6

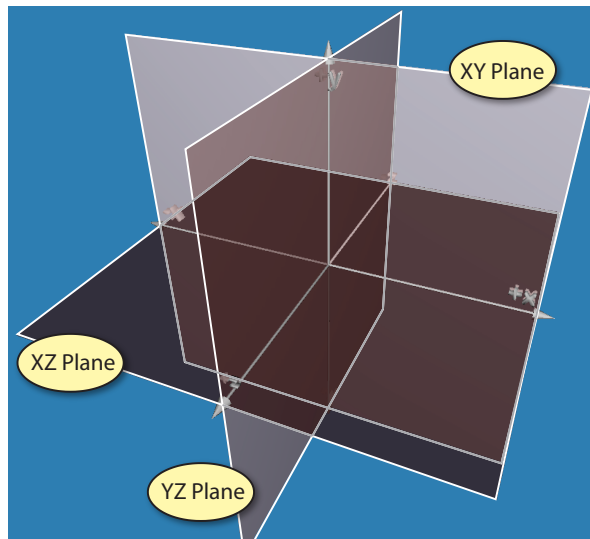
Planes in 3D Space



Three special planes exist. These planes are infinite in size and pass through all the points on two of the axes with the third axes value always zero. The planes are known by the axes they pass through; they are the XY plane, the XZ plane and the YZ plane (see FIG-23.7).

FIG-23.7

The XY, XZ and YZ
Planes



When measuring the angle of an item such as a line in 3D space we would measure it relative to one or more of these planes.

Modelling Ideas and Terminology

Now that we've had a look at the basic geometry of 3D space, we are going to move on to explain some of the jargon you are likely to come across when dealing with 3D models.

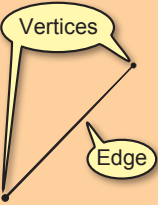
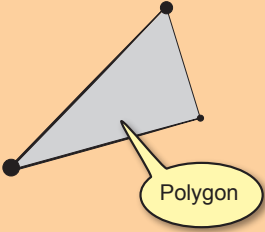
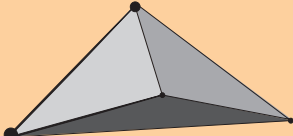

Elements of a Model

At the most fundamental level, 3D models are constructed by defining points in space (known as **vertices**). Joining two vertices creates an **edge** and joining three or four edges produces a **polygon**. The simplest polygon is the triangle formed by three

edges. A collection of polygons is known as a **polygonal mesh** and it is this mesh that defines the complete 3D model. The basic idea behind this construction is shown in FIG-23.8.

FIG-23.8

Elements of a 3D Shape

<p>If we take two vertices and join them together, we create an edge.</p>	<p>If we create a third vertex and two more edges, then we have created a polygon.</p>
	
<p>By creating several more linked polygons, we produce a polygonal mesh.</p>	<p>In reality, neither the vertices nor edges are actually visible in the final model. Notice that the faces of the model are shaded to create a 3D effect.</p>
	

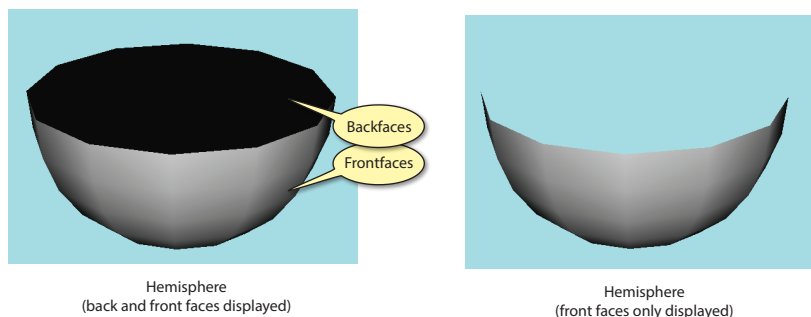
Front and Back Faces

Like a coin, every polygon has two sides or faces: one face is designed to be seen by the viewer (known as the **front face**) and one which is normally hidden (the **back face**). In the pyramid above, the back face of each polygon that makes up the shape is on the inside of the pyramid and hence can never be seen.

The default drawing mode for most game engines is to **cull** (not draw) back faces. This isn't usually a problem since, like the pyramid, back faces occur where they can never be viewed. However, some models leave back faces exposed. For example, if we were to display half a sphere, then the back faces of its polygons would be exposed (see FIG-23.9).

FIG-23.9

Culling Backfaces



As you can see from FIG-23.9 some models require the back faces to be displayed if the object is to be displayed accurately.

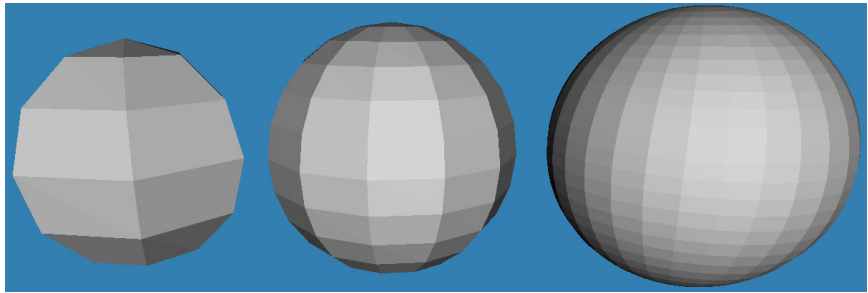
Model Resolution

In general, the more polygons a shape contains the more realistic the model will appear. The number of polygons in a model is usually referred to as the **polycount**.

The downside to an increased polygon count is the increase in processing required by the Graphics Processing Unit (GPU). Increase the polycount too much and the frame rate of your game will fall. FIG-23.10 shows three versions of a sphere, each with an increased polycount.

FIG-23.10

The Effect of Increasing Polycounts

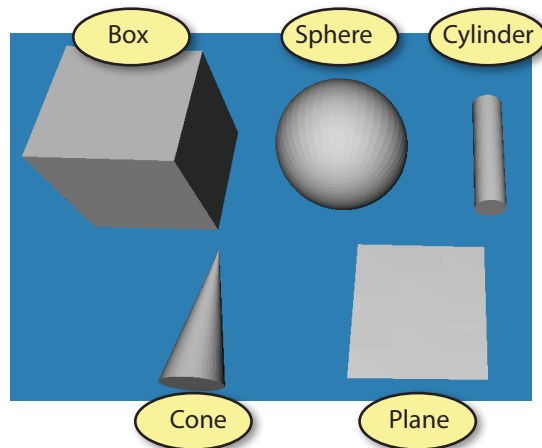


3D Primitives

When you use a 3D modelling package such as Milkshape to create a model, you normally start with a basic shape such as a box, sphere, or cylinder. These are known as **3D primitives**. Each modelling package offers a slightly different set of primitives. A set of typical primitives is shown in FIG-23.11.

FIG-23.11

3D Primitives



To create other shapes, primitives may be added together or the vertices of a primitive can be moved, merged, increased, or deleted.

AGK has commands to create box, sphere, cylinder, cone and plane primitives as well as a command to load existing models stored in the .OBJ format. Many 3D design packages (including Milkshape) can output models in this format.

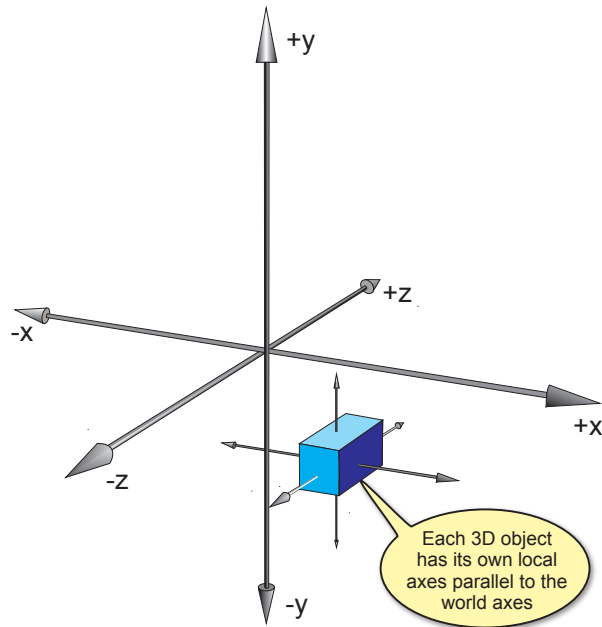
When creating 3D primitives in AGK, they are automatically centred at the origin, while loaded models are positioned corresponding to their original position in the modelling package.

Local Axes

When an object is added to 3D space, it is automatically assigned a set of **local axes** that have their origin at the centre of that object (see FIG-23.12).

FIG-23.12

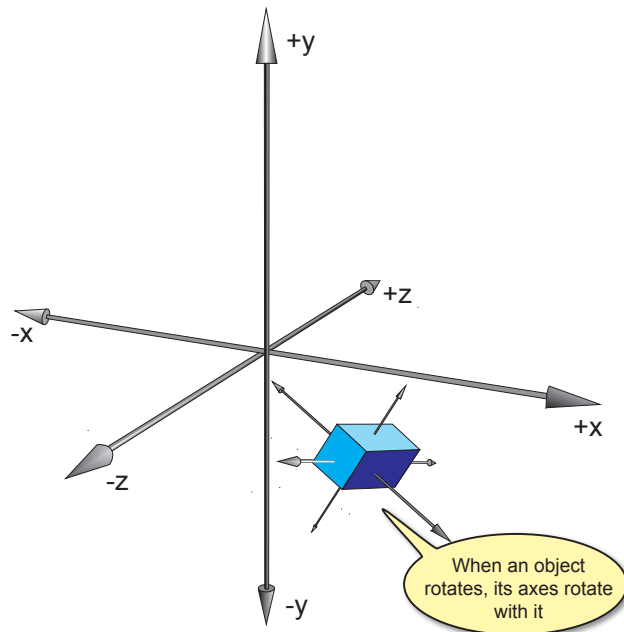
Local Axes



When an object is rotated, that rotation is normally relative to the object's own local axes, but the axes rotate along with the object itself (see FIG-23.13).

FIG-23.13

Local Axes Rotate with their Object



Notice that once an object has been rotated, the local axes are no longer aligned with the world axes.

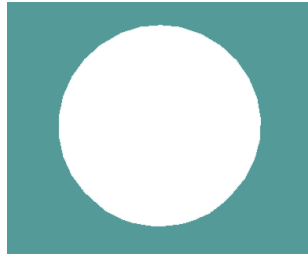
Lighting

One of the secrets of a realistic 3D model is good lighting. With the correct lighting effects, not only can you add to the realism of your model, but you can also create various atmospheric tones such as a creepy, dark, ill-light corridor, a bright sun on a desert landscape, flashing lights at an accident, the flickering light of a fire, or any other effect you wish.

AGK 3D scenes default to **ambient light**. Ambient light is a light in which the rays of light are coming equally from all directions. Ambient light on its own produces a rather unsatisfying flattening effect. For example, the sphere shown in FIG-23.14 has no shadowed area when viewed under only ambient light and hence the whole 3D effect is lost.

FIG-23.14

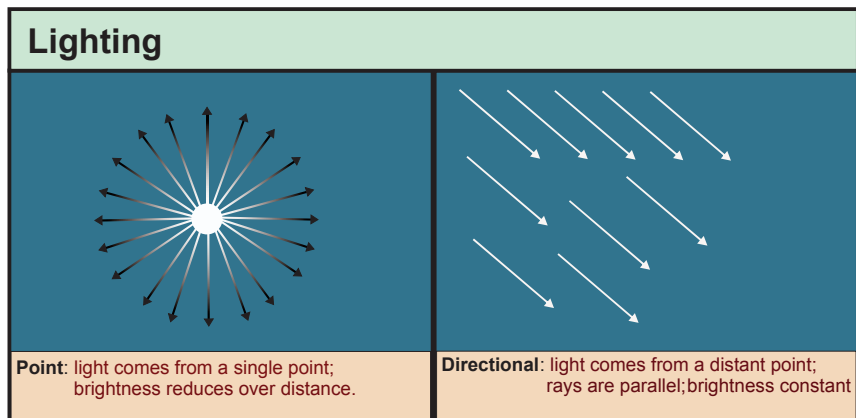
A Sphere Under
Ambient Light



To produce other lighting effects in your 3D scene, you must create and position the additional lights you require. AGK offers two additional basic types of additional lighting: **point light** and **directional light**. We might compare these two lighting types to a real world naked bulb (point lighting) and sunlight (directional light) (see FIG-23.15).

FIG-23.15

Point and Directional
Lights

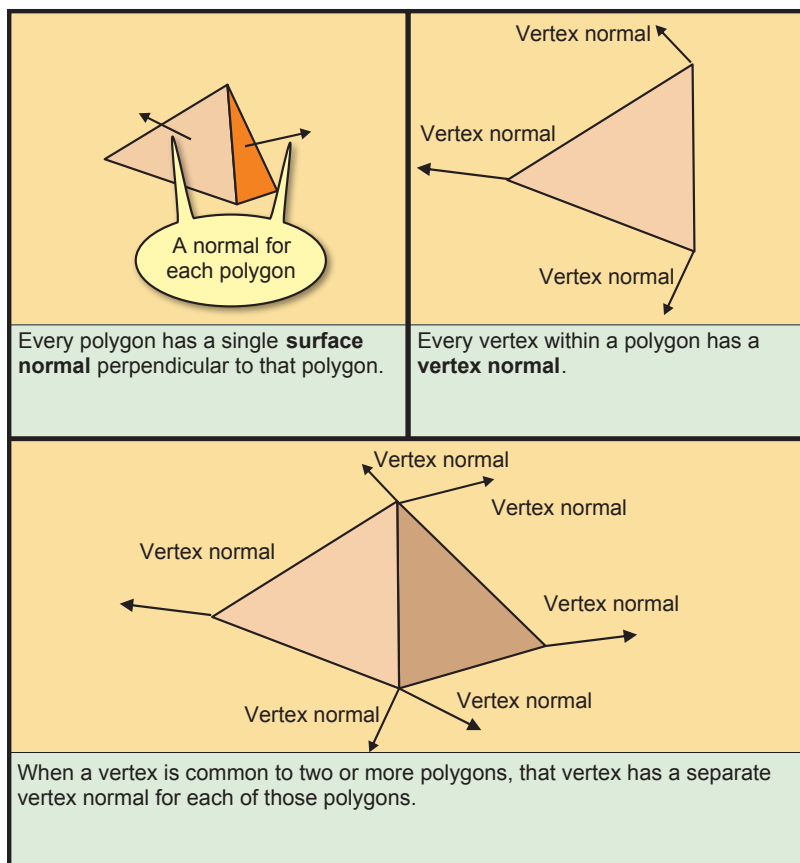


Normals

Calculating the shadows and highlights on a model created by the lighting is a complex task, but the GPU is aided in this task by a set of vectors know as **normals**. We can think of a normal as an invisible line radiating from points on each of the polygons that go to make up a model. There are two types of normals: **surface normals** and **vertex normals**. Every polygon within a model has a single surface normal which is perpendicular to the polygon's surface. Each vertex of a polygon has a vertex normal (so there are three vertex normals on a triangular polygon). Each vertex normal is perpendicular to the two edges that meet at that point (see FIG-23.16).

FIG-23.16

Surface and Vertex Normals

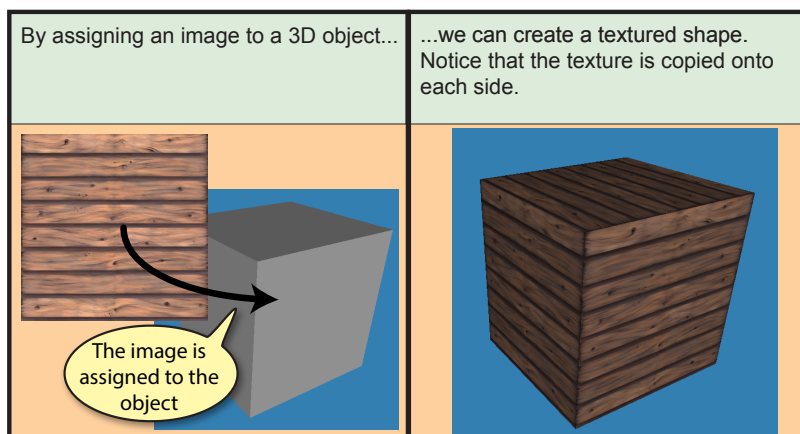


Texturing

When you first create a 3D object, its surfaces will be displayed in boring shades of grey. To add realism, making the basic shape take on the look of the object it is meant to represent, we can wrap an image around those surfaces. This is known as **texturing**. FIG-23.17 gives an overview of the process.

FIG-23.17

Texturing an Object

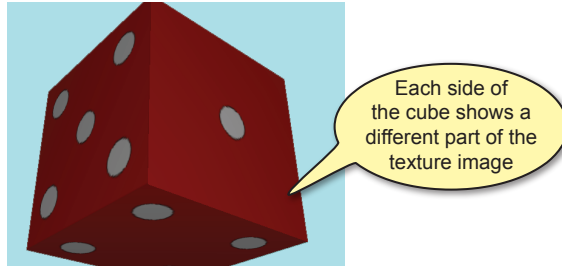


You may recall that when an image was added to a sprite, the image was assumed to have a width and height of 1 no matter what the actual dimensions of the image. The coordinates of the image are referred to as UV coordinates rather than XY coordinates.

A record of the UV coordinates of the image that are to be mapped to each of the vertices in the model is used when creating the textured model. Although AGK allows you to add an image texture to a model, you have no control over how that image is mapped onto the surface of the model. This is fine if you want to create something like a wooden box where you are happy to have the same image on each side of the box. However, if you want greater control over the placement of the image, you must create and texture the model within a 3D modelling package and import the resulting model and image used into your AGK program. For example, the cube in FIG-23.18 could easily be created from within AGK, but because each side of the box has a different texture, the model needs to be set up using a modelling package such as Milkshape.

FIG-23.18

Mapping a Texture



In AGK you must assign an image to a 3D model to create a textured effect. Even when the texture has been set up within a modelling package, you still need to include AGK statements within your app to load and assign the image being used by the model.

Animation

Although simple animation can be achieved by moving, rotating or resizing an object, true animation - where parts of a single object move in relation to each other - has to be created with the 3D modelling package being used.

This is normally achieved by first creating the basic object and then adding a skeleton to that model. A skeleton consists of bones and joints. The joints are then linked to specific groups of vertices. When the joints of a skeleton are moved, the associated vertices also move; stretching and compressing the associated polygons.

The animation is created as a sequence of frames. The modeller sets up specific frames within the animation (known as key frames) by moving the joints of the skeleton and the modelling package automatically calculates the skeleton and polygon positions for intermediate frames (see FIG-23.19).

FIG-23.19

Creating a 3D Animation

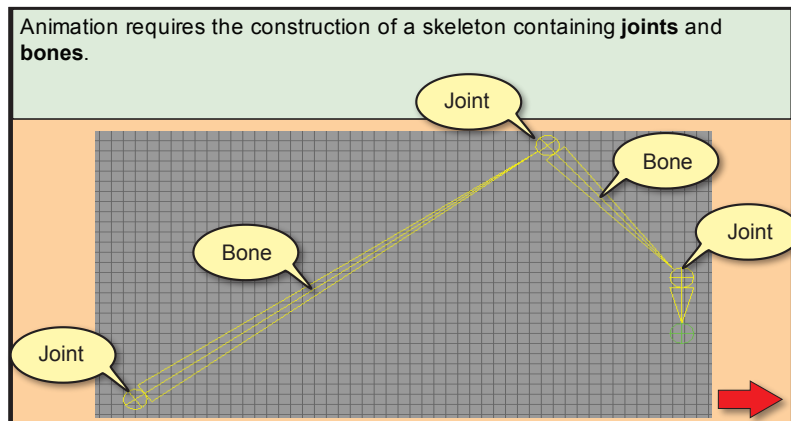
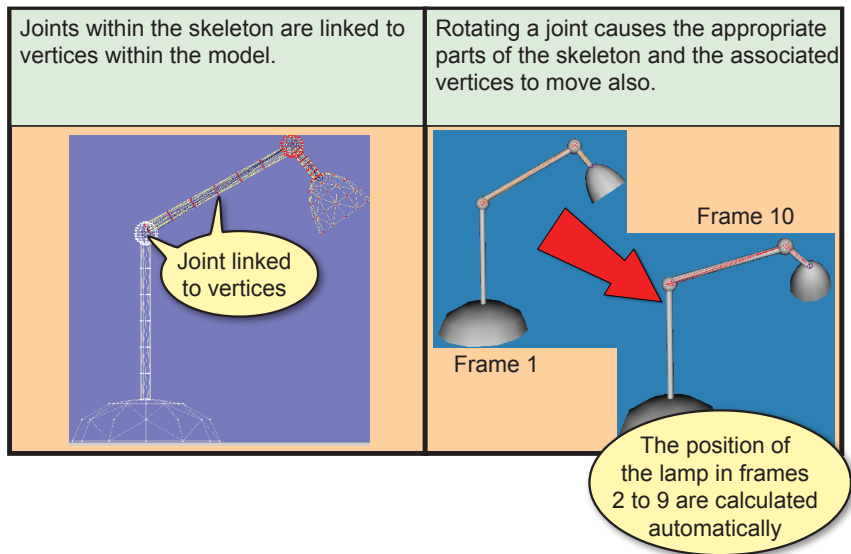


FIG-23.19
(continued)

Creating a 3D
Animation



Currently, AGK does not offer commands to play 3D animation files.

Shaders

A shader is a piece of software executed by a device's Graphics Processing Unit (GPU) hardware to calculate the shading to be displayed on the 3D objects that are displayed on the screen. But shaders can also create special effects such as distorting an object's shape, colour, transparency or surface smoothness.

In AGK you can load up your own shader software (which you have to create separately) to change the look of the images produced on the screen.

AGK also allows you to add up to 7 additional images to a 3D object (other than the one used for basic texturing) which can be used by the shader to affect the appearance of that object.

Cameras

What you see on the screen is only a part of the 3D world you create. It's a bit like watching a news item on TV; what you see is the part of the surroundings captured by the TV camera. Move the camera to a different position and you get a different perspective of the same world.

In AGK a camera is created automatically. It is initially placed in a position roughly equivalent to the eyes of the viewer, slightly raised on the y-axis and moved along the negative part of the z-axis. AGK commands allow you to reposition and rotate the camera as well as select the position at which the camera is pointing and to zoom in and out.

Summary

- 3D space is measured using x, y and z axes.
- These main axes are known as the world axes.
- The z-axis in AGK space has its positive end facing into the screen. This is

known as the left-handed coordinate system.

- A point in 3D space is defined using three values (x, y, z) representing its displacement along each axes.
- A line is defined by the coordinates of its two end points.
- The offset values of a line are calculated by subtracting the start point from the end point for all three values (x, y and z).
- A 3D vector is defined by its offset values in all three directions (x, y and z).
- The length of a line, or the magnitude of a vector is calculated as the square root of the sum of the squares of the three offsets.
- A plane is a flat 2D surface within 3D space.
- The main planes are the XY plane, the XZ plane and the YZ plane.
- 3D objects are constructed from polygons.
- The corner points on a polygon are known as vertices.
- The line between two vertices is known as an edge.
- Polygons have a front face and a back face.
- Normally, back faces are not displayed on the screen.
- Generally, the more polygons used in the construction of an object, the more accurate the representation.
- The greater the number of polygons used in a scene, the more work the graphics processing unit (GPU) has to perform.
- Using a high number of polygons can lead to a reduced frame rate.
- 3D models are constructed by joining or manipulating primitive shapes.
- Typical primitive shapes are: the box, sphere, cylinder, cone and plane.
- Object transformation involves moving, resizing or rotating.
- Each object has its own local set of axes.
- Local axes move and rotate with the object to which they belong.
- Lighting is used to create highlights and shadows.
- AGK uses three types of lighting: ambient, directional and point.
- Ambient light is light which comes equally from all directions. Ambient light causes neither highlights or shadows and gives an object a flat appearance.
- Directional light represents light arriving from an infinite distance with all rays parallel and their brightness remaining constant over distance. Typically, this is used to represent sunlight.
- Point light. A point light originates from a specific point and radiates equally in all directions. The light diminishes over distance. Typically, this is used to represent a light bulb.
- A polygon uses vertex and surface normals to help in the calculation of shading.
- A image can be mapped onto a 3D object to create a surface texture for that

object.

- Simple texturing is carried out automatically by AGK; more complex texturing must be applied within a 3D modelling package.
- A shader is a piece of software executed by the GPU to modify the appearance of the final image appearing on the screen.
- What appears on the screen when using 3D models is the view created by a virtual camera.
- The virtual camera can be moved, rotated, made to point at a specific location and used to zoom in and out.

Creating a First 3D App

Introduction

There are many new commands in AGK for handling 3D. And although these can be logically grouped into sections on primitives, texturing, lighting and cameras, etc., to get a good overview of the process we need to start by looking at one or two commands from these groups to create our first app.

Statements

LoadObject()

If you have created a 3D object using a modelling package, you can use the `LoadObject()` statement to load that model into your app. The `LoadObject()` statement has the format shown in FIG-23.20.

FIG-23.20

`LoadObject()`

Format 1

`LoadObject` ((`id` , `file` [(, `scale`]))

Format 2

`integer` `LoadObject` ((`file` [(, `scale`]))

where

id	is an integer value giving the ID to be assigned to the new object.
file	is a string giving the name of the file containing the 3D model. The file must be in OBJ format and held in the project's <i>media</i> folder.
scale	is a real value giving the height of the object. Other dimensions are scaled appropriately.

The model will be positioned at a spot corresponding to its original position in the modelling package when it was saved to a file. Most models are created with their centre at the origin.

The program in FIG-23.21 loads and displays a model which contains a labelled representation of the three axes.

FIG-23.21

Loading and Displaying
a Model

```
rem *** Loading a displaying an existing model ***  
  
rem *** Load model ***  
LoadObject(1,"Axes.obj",15)  
  
rem *** Display model ***  
do  
    Sync()  
loop
```


Activity 23.2

Start a new project called *First3D*. Enter the code given in FIG-23.21 then compile the program.

Copy the file *AGKDownloads/Chapter23/Axes.obj* into the project's *media* folder. In the project's *setup.agc* file, change the dimensions of the app window to 900 x 900.

Run the program and observe the results then save your project.

The model displayed by *First3D* doesn't look all that clear. This is a combination of the camera's position and the default lighting used.

SetCameraPosition()

By default, the virtual camera, which creates the image you see on the screen when in 3D mode, is placed at position (0,10,-20). You can reposition it using `SetCameraPosition()` (see FIG-23.22).

FIG-23.22

SetCameraPosition()

SetCameraPosition ((id , x , y , z))

where

id is an integer value giving the ID of the camera. Currently, AGK contains no commands which allow you to create additional cameras so this value is always 1 (the ID of the default camera).

x, y, z are real values giving the coordinates at which the camera is to be placed.

Activity 23.3

Modify *First3D*, moving the camera to position (20,10,-20). Run the program and observe the results. What part of the axes model is visible?

Modify your program so that the camera moves from its starting position (0,10,-20) at a speed of 0.1 units along the x-axis every 50 milliseconds until it arrives at the required position of (20,10,-20).

Run and save your program.

SetCameraLookAt()

The result from Activity 23.3 highlights that, not only do we need to be able to reposition a camera, but we also need to make the camera point in a specific direction so that the part of the world we want to see appears on the screen. We can point the "lens" of the camera at a particular position using the `SetCameraLookAt()` statement (see FIG-23.23).

FIG-23.23

SetCameraLookAt()

SetCameraLookAt ((id , x , y , z , roll))

where

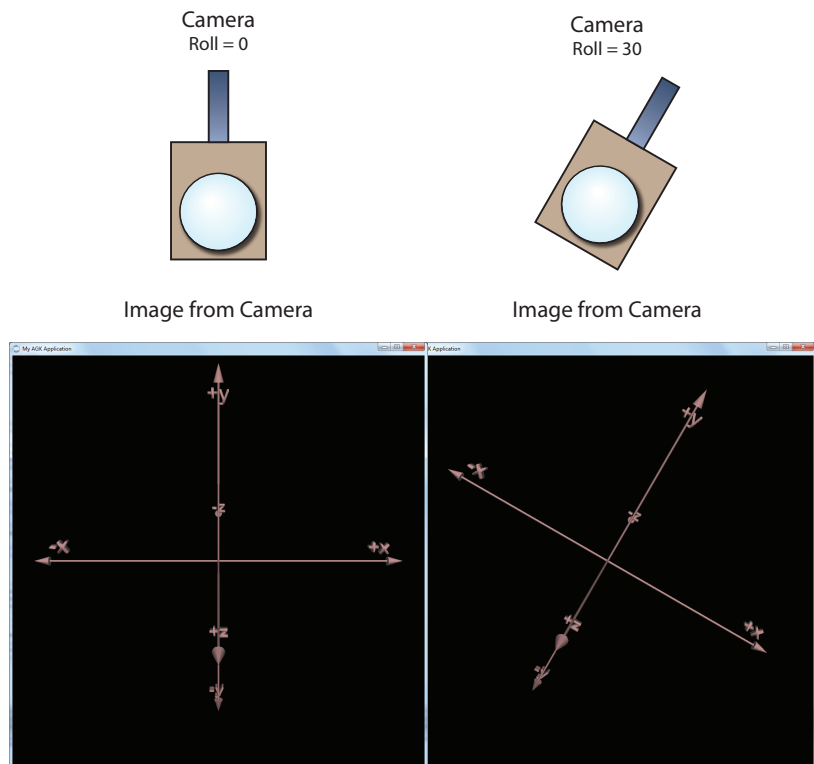
- id** is an integer value giving the ID of the camera. Currently, AGK contains no commands which allow you to create additional cameras so this value is always 1 (the ID of the default camera).
- x, y, z** are real values giving the coordinates at which the camera lens is to be pointed. This point will be at the centre of the screen.
- roll** is a real number giving the angle of the camera roll. Using a value other than zero gives the effect of a camera tilted to the side. That is, the camera is rotated about its own z-axis.

This command should be called each time the camera is moved if you want it to remain pointing at a particular spot.

The effect of using the *roll* parameter is demonstrated in FIG-23.24.

FIG-23.24

Using the Camera Roll Option



Activity 23.4

Modify *First3D* so that the camera is always pointing at the origin as it moves to position (20,10,-20).

Run and save your program.

CreateLightDirectional()

The default ambient light is harsh and provides little or no shading to give the models a more realistic look. One way to enhance the shading is to add your own directional light. This will give the effect of sunlight on the model.

FIG-23.25

CreateLightDirectional() To create a directional light, use `CreateLightDirectional()` (see FIG-23.25).

`CreateLightDirectional (id , x , y , z , ir , ig , ib)`

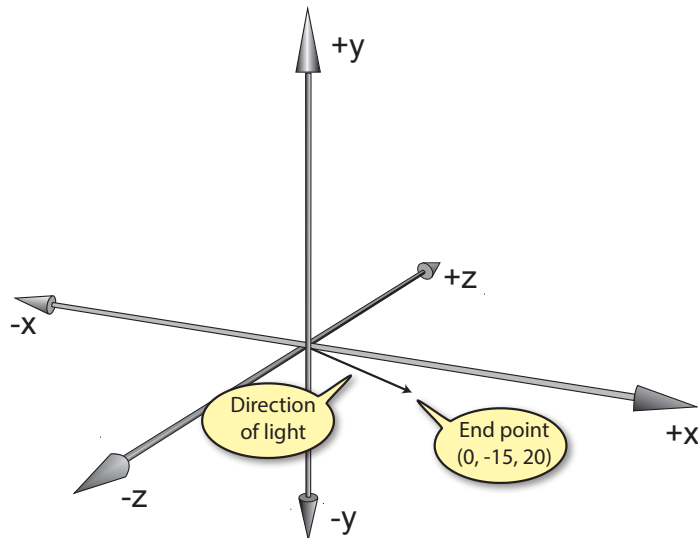
where

id	is an integer value giving the ID to be assigned to the point light.
x, y, z	are real values giving the vector of the light rays.
ir, ig, ib	are integer values (0 to 255) giving the intensities of the red, green and blue components of the colour of the light emitted. Set all three to 255 for white light.

To help calculate the displacement values for x , y , and z , imagine a line drawn from the origin which is parallel to the rays of light you want to create; x , y and z are the coordinates of the end point of that line (see FIG-23.26).

FIG-23.26

Calculating the Light's Vector



For example, to create a red-tinted light in the direction indicated in the diagram, we would use the line

```
CreateLightDirectional (1,0,-15,20,200,150,150)
```

Activity 23.5

Modify *First3D* so that a directional light is added immediately after the axes model is loaded.

Run and save your program.

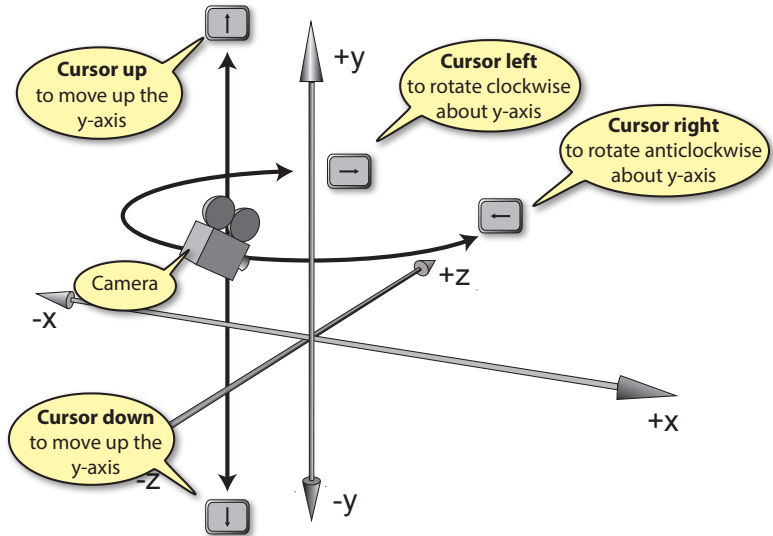
User Control of the Camera

Although the camera changes position in project *First3D*, the user has no control over that movement. It would be more useful if we could control camera movement from the mouse, touch screen or keyboard.

The next program allows the user to control camera movement. The camera starts at position (0,10,-25) but the left and right cursor keys of the keyboard can be used to rotate the camera about the y-axis while the up and down arrow keys move the camera up and down in the y direction. The camera is always pointed towards the origin. The movement options for the camera are shown in FIG-23.27.

FIG-23.27

Camera Movement
Options



This time all the camera information is stored in a global record structure and the camera's movement is control by a separate function, *HandleCamera()*. The program code is given in FIG-23.28.

FIG-23.28

User-Controlled Camera
Movement

```
rem ***User Controlled Camera Movement ***

rem *** Structure for camera details ***
type CameraDataType
  x as float
  y as float
  z as float      // Camera's coords
  dist as float   // Camera's distance from y-axis
  angle as float  // Camera's rotation about y-axis
endtype

rem *** Global Variable ***
rem *** Camera info ***
global camera as CameraDataType
camera.x = 0
camera.y = 10
camera.z = -25
camera.dist = 25
camera.angle = -90
```



FIG-23.28

(continued)

User-Controlled Camera
Movement

```

rem *** Load model ***
LoadObject(1,"Axes.obj",20)

rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,200,150,150)
rem *** Add text to show camera's coords ***
CreateText(1,"")
CreateText(2,"")
SetTextPosition(2,0,6)
CreateText(3,"")
SetTextPosition(3,0,12)

rem *** Give user camera control ***
do
    HandleCamera()
    rem *** Show camera position ***
    SetTextString(1,"X: "+Str(camera.x,1))
    SetTextString(2,"Y: "+Str(camera.y,1))
    SetTextString(3,"Z: "+Str(camera.z,1))
    rem *** Update screen ***
    Sync()
loop

function HandleCamera()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            case 37: //left cursor, decrease angle
                dec camera.angle
            endcase
            case 38: //up cursor, increase y
                camera.y=camera.y+0.25
            endcase
            case 39: //right cursor, increase angle
                inc camera.angle
            endcase
            case 40: //down cursor, decrease y
                camera.y=camera.y-0.25
            endcase
        endselect
    endif
    rem *** Calculate new x and z coordinates ***
    camera.x = camera.dist*cos(camera.angle)
    camera.z = camera.dist*sin(camera.angle)
    rem *** Reposition camera to match ***
    SetCameraPosition(1,camera.x,camera.y,camera.z)
    rem *** Make camera point at origin ***
    SetCameraLookAt(1,0,0,0,0)
endfunction

```

Activity 23.6

Start a new project called *Second3D*, which implements the code given in FIG-23.28. Copy *Axes.obj* to the project's *media* folder. Set screen size to 900 x 900.

Run the program and use the cursor keys to move the camera about the 3D space. Save your program.

Activity 23.7

Modify *Second3D* so that the camera's roll increments by 1° when the + key is pressed (code 187) and decrements by 1° when the - key is pressed (189).

The roll details should be added as a new field within the *CameraDataType* structure.

Test the new feature then save your project.

Summary

- Use `LoadObject()` to load an existing 3D model.
- Only .OBJ models may be loaded.
- Use `SetCameraPosition()` to position the virtual camera.
- Use `SetCameraLookAt()` to make the camera lens point at a particular point in 3D space.
- `SetCameraLookAt()` can also be used to roll the camera about its z-axis.
- By default, AGK uses ambient light which creates a flat appearance with no shadows.
- Use `CreateDirectionalLight()` to create a directional light.
- A directional light will create shadows within the model.

Object Creation and Modification

Creating Primitives

For simple games you may not need to create complex 3D shapes using a modelling package. Instead, you may be able to make do with the basic primitive shapes that can be created directly using AGK commands.

CreateObjectBox()

To create a box, use the `CreateObjectBox()` statement (see FIG-23.29).

FIG-23.29

CreateObjectBox()

Format 1

`CreateObjectBox` (`id` , `w` , `h` , `d`)

Format 2

integer `CreateObjectBox` (`w` , `h` , `d`)

where

- | | |
|-----------|---|
| id | is an integer value giving the ID to be assigned to the new object. |
| w | is a real value giving the width of the object. |
| h | is a real value giving the height of the object. |
| d | is a real value giving the depth of the object. |

Format 1 of the statement allows you to specify the ID to be assigned to the object; format 2 will assign an ID automatically and return that ID.

Activity 23.8

Modify *Second3D* so that a box is added to the project. The box should have an ID of 2 and be 3 units wide, 2 high and 1 deep. Place the appropriate AGK statement immediately before the `do..loop` section.

Run the program and use the cursor keys to move the camera about the 3D space. Where is the centre of the box positioned?

Save your program.

Primitives created using AGK statements are always centred on the origin.

CreateObjectSphere()

A polygon-based curve is always an approximation of a true curve. The more polygons assigned to the model, the nearer it will come to looking like a true curve, but the price you pay for this is larger storage and greater processing requirements which can reduce the frame rate of a game. Of course, the most demanding of

primitive shapes is the sphere, since all parts of its surface are curved.

To create a sphere in AGK, use `CreateObjectSphere()` (see FIG-23.30).

FIG-23.30

`CreateObjectSphere()`

Format 1

`CreateObjectSphere (id , d , r , c)`

Format 2

integer `CreateObjectSphere (d , r , c)`

where

id	is an integer value giving the ID to be assigned to the new object.
d	is a real value giving the diameter of the sphere.
r	is an integer value giving the number of rows used when constructing the sphere.
c	is an integer value giving the number of columns used when constructing the sphere.

Increasing the number of rows and columns will improve the realism of the sphere.

Format 1 of the `CreateObjectSphere()` statement allows you to specify the ID to be assigned to the object; format 2 will assign an ID automatically and return that ID.

Activity 23.9

Modify *Second3D* replacing the box with a sphere. Set the diameter of the sphere to 5 with 4 rows and 4 columns and observe the accuracy of the actual shape created.

Modify the code so that 20 rows and 20 columns are used. How does this affect the accuracy of the sphere? Save your program.

CreateObjectCone()

To create a cone, use the `CreateObjectCone()` statement (see FIG-23.31).

FIG-23.31

`CreateObjectCone()`

Format 1

`CreateObjectCone (id , h , d , s)`

Format 2

integer `CreateObjectCone (h , d , s)`

where

id	is an integer value giving the ID to be assigned to the new object.
-----------	---

h	is a real value giving the height of the cone.
d	is a real value giving the diameter of the cone's base.
s	is an integer value giving the number of segments used when constructing the cone.

Increasing the number of segments will increase the realism of the cone.

Format 1 of the statement allows you to specify the ID to be assigned to the object; format 2 will assign an ID automatically and return that ID.

Activity 23.10

Modify *Second3D* replacing the sphere with a cone. Set the height to 6 and the diameter of the base to 2.5.

On the first run, use a 5 segment cone; on the second run, use a 15 segment cone and observe the difference between the two models.

Save your program.

CreateObjectCylinder()

To create a cylinder use `CreateObjectCylinder()` (see FIG-23.32).

FIG-23.32

CreateObjectCylinder()

Format 1

`CreateObjectCylinder (id , h , d , s)`

Format 2

`integer CreateObjectCylinder (h , d , s)`

where

id	is an integer value giving the ID to be assigned to the new object.
h	is a real value giving the height of the cylinder.
d	is a real value giving the diameter of the cylinder's base.
s	is an integer value giving the number of segments used when constructing the cylinder.

Format 1 of the statement allows you to specify the ID to be assigned to the object; format 2 will assign an ID automatically and return that ID.

Activity 23.11

Modify *Second3D* replacing the cone with a cylinder. Set the height to 4 and the diameter of the base to 3. Use 12 segments in the construction.

Run and save your program.

CreateObjectPlane()

Use `CreateObjectPlane()` to create a 2D plane. The statement has the format shown in FIG-23.33.

FIG-23.33

CreateObjectPlane()

Format 1

CreateObjectPlane ((id , w , h))

Format 2

integer CreateObjectPlane ((w , h))

where

id is an integer value giving the ID to be assigned to the new object.

w is a real value giving the width of the plane.

h is a real value giving the height of the plane.

The plane created will be part of the XY plane.

Format 1 of the statement allows you to specify the ID to be assigned to the object; format 2 will assign an ID automatically and return that ID.

Activity 23.12

Modify *Second3D* adding a plane 8 units wide and 7 units high.

Run your program. How do the plane and cylinder interact?

Can the plane be viewed from both sides?

Save your program.

CloneObject()

You can make a copy of an existing 3D object using the `CloneObject()` statement (see FIG-23.34).

FIG-23.34

CloneObject()

CloneObject ((id , objid))

where

id is an integer value giving the ID to be assigned to the new object.

objid is an integer value giving the ID of the existing object being copied.

Cloning an object makes a separate copy of an existing object. The copy can then be manipulated independently of the original.

InstanceObject()

You can also duplicate an existing object using the `InstanceObject()` statement. However, using this command, the original object and the copy continue to share vertex data. This does not stop you performing independent changes to the copy or even changing the texture used, but the copy remains linked to the original object via the shared data so deleting the original would cause the copy to be deleted or your program to crash. The `InstanceObject()` statement's format is given in FIG-23.35.

FIG-23.35

InstanceObject()

`InstanceObject (id , objid)`

where

id is an integer value giving the ID to be assigned to the new object.

objid is an integer value giving the ID of the existing object being copied.

The program in FIG-23.36 highlights the similarities of cloned and instanced objects but also demonstrates that instanced objects rely on the existence of the original.

FIG-23.36

Duplicating Objects

```
rem *** Duplicating Objects ***

rem *** Create cube ***
CreateObjectBox(1,5,5,5)

rem *** Clone object ***
CloneObject(2,1)
SetObjectPosition(2,-6,0,0)

rem *** Instance Object ***
InstanceObject(3,1)
SetObjectPosition(3,6,0,0)

rem *** Create Directional light ***
CreateLightDirectional(1,10,10,10,255,255,255)

rem *** Position and point camera ***
SetCameraPosition(1,0,10,-30)
SetCameraLookAt(1,0,5,0,0)

ResetTimer()
state = 0
do
    rem *** Delete original object after 3 secs ***
    if Timer() > 3 and state = 0
        DeleteObject(1)
        state = 1
    endif
    Sync()
loop
```

Activity 23.13

Create a new program called *Copy3D* and implement the code in FIG-23.36.

Test and save your program.

GetObjectExists()

To check if a 3D object of a given ID currently exists, use `GetObjectExists()` (see FIG-23.37).

FIG-23.37

GetObjectExists()

integer `GetObjectExists (id)`

where

id is an integer value giving the ID to be checked.

The function returns 1 if the object exists, otherwise zero is returned.

DeleteObject()

An existing 3D object can be deleted using `DeleteObject()` (see FIG-23.38).

FIG-23.38

DeleteObject()

`DeleteObject (id)`

where

id is an integer value giving the ID of the object to be deleted.

Activity 23.14

Modify *Second3D* so that the cylinder is deleted if the *delete* key (code 46) is pressed. This should be done using a second function named *HandleCylinderDelete()* which has a similar structure to *HandleCamera()*.

Run and save your program.

Object Appearance

We can change the appearance of a 3D object in various ways. Perhaps the simplest is to change the surface colour. Another option is to texture the object's surface using an image.

SetObjectColor()

To set the surface colour of an object, use `SetObjectColor()` (see FIG-23.39).

FIG-23.39

SetObjectColor()

`SetObjectColor (id , ir , ig , ib , it)`

where

id is an integer value giving the ID of the object.

ir is an integer value (0 to 255) giving the intensity of the red component within the colour.

ig is an integer value (0 to 255) giving the intensity of the green component within the colour.

ib is an integer value (0 to 255) giving the intensity of the blue component within the colour.

it is an integer value (0 to 255) giving the opacity of the colour (0: fully transparent; 255: fully opaque).

Activity 23.15

Reload *Second3D*. Modify the directional light so that its colour components are 250, 250, 250.

Set the colour of the plane to 90, 150, 200 with transparency set to 255. Run and save your program.

SetObjectImage()

To texture an object with an image file, use `SetObjectImage()` (see FIG-23.40).

FIG-23.40

`SetObjectImage()`

`SetObjectImage ((id , imgid , istg)`

where

id is an integer value giving the ID of the object.

imgid is an integer value giving the ID of a previously loaded image which is to be used when texturing the object.

istg is an integer value (0 to 7) giving the stage at which the image is to be applied. Use stage 0 for a standard texture.

More than one image can be applied to the object, but images applied to stages 1 to 7 are used by the shader to create other effects and are not normally visible.

Activity 23.16

Modify *Second3D*, deleting the cylinder and plane objects and replacing them with a box measuring 6 x 6 x 6. Load the image *Wood.png* into the project's *media* folder. Load the image and apply it to the surface of the box. Run and save your program.

The `HandleCylinderDelete()` function and the call to it can also be deleted from the program.

SetObjectTransparency()

When you make an object translucent using `SetColorObject()` or when you use a texture image which contains transparent elements, you need to call `SetObjectTransparency()` in order to activate the transparency.

To set an object's transparency mode, use `SetObjectTransparency()` (see FIG-23.41).

FIG-23.41

`SetObjectTransparency()`

`SetObjectTransparency ((id , it)`

where

id is an integer value giving the ID of the object.

it is an integer value (0, 1 or 2) giving the transparency

setting. (0: opaque, 1: transparency activated, 2: additive blending).

To show how this command affects the appearance of objects on screen, the program in FIG-23.42 uses a plane positioned on the axes. The plane is first textured with an image consisting of an opaque yellow square with a hole near its centre. Initially, the image has transparency switched off. Pressing the *return* key toggles the transparency setting. Pressing *2* changes the image used to one in which the yellow of the square is translucent; pressing *1* returns to the original image.

FIG-23.42

The Effect of
Transparency

```
rem *** Transparency ***

rem *** Object Attributes Type ***
type ObjectAttributesType
    image as integer
    transparency as integer
endtype

rem *** Global Variable ***
global ObjAttr as ObjectAttributesType
objAttr.image = 1
objAttr.transparency = 0

rem *** Load axes ***
LoadObject(1,"Axes.obj",20)
rem *** Create plane ***
CreateObjectPlane(2,12,12)
rem *** Add Images used ***
LoadImage(1,"SquareWithHole.png")
LoadImage(2,"TransparentSquare.png")

rem *** Set up light ***
CreateLightDirectional(1,10,-10,20,200,200,200)
rem *** Position camera ***
SetCameraPosition(1,10,10,-20)
SetCameraLookAt(1,0,0,0,0)

rem *** Allow user to switch transparency mode and image ***
do
    HandlePlaneAttributes()
    Sync()
loop

function HandlePlaneAttributes()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key just pressed, process it***
    if GetRawKeyPressed(key)=1
        select key
            case 13: //Enter key, toggle transparency
                objAttr.transparency = 1 - objAttr.transparency
            endcase
            case 49: //1 key, toggle image
                objAttr.image = 3 - objAttr.image
            endcase
        endselect
        SetObjectImage(2,objAttr.image,0)
        SetObjectTransparency(2,objAttr.transparency)
    endif
endfunction
```

Activity 23.17

Start a new project called *Transparency3D* and implement the code given in FIG-23.42. Copy the files *Axes.obj*, *SquareWithHole.png* and *TransparentSquare.png* into the project's *media* folder.

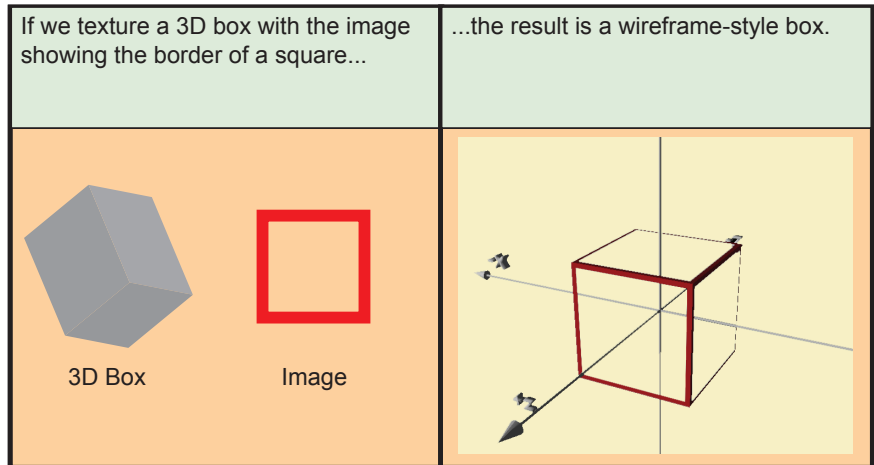
Run your program and observe the results as you vary the image and transparency setting used.

Save your program.

By using an appropriate image, you can create shapes that would normally require a lot more effort at the modelling stage (see FIG-23.43).

FIG-23.43

Modifying a Shape using an Image



Activity 23.18

Modify *Second3D* to display a cube (6 units in each direction) which has been textured using the image *Perimeter.png*. Remember to copy the image to the *media* folder.

Observe the result produced then save your program.

GetObjectTransparency()

To discover the transparency setting of a specific object, use the `GetObjectTransparency()` statement (see FIG-23.44).

FIG-23.44

`GetObjectTransparency()` integer `GetObjectTransparency` (`id`)

where

id is an integer value giving the ID of the 3D object whose transparency setting is to be found.

The value returned (0, 1, or 2) gives the transparency setting assigned by any previous `SetObjectTransparency()` statement for that object. The default setting is zero.

SetObjectVisible()

Irrespective of the transparency setting, an object can be made invisible (or returned to normal) using the `SetObjectVisible()` statement (see FIG-23.45).

FIG-23.45

SetObjectVisible()

`SetObjectVisible (id , iv)`

where

id is an integer value giving the ID of the object.

iv is an integer value (0 or 1) which makes an object invisible (0) or visible (1). 1 is the default value for all objects.

GetObjectVisible()

To determine if an object is currently invisible, use `GetObjectVisible()` (see FIG-23.46).

FIG-23.46

GetObjectVisible()

integer `GetObjectVisible (id)`

where

id is an integer value giving the ID of the object.

The function returns the visibility setting of the specified object. If the object is visible, 1 is returned; if invisible, zero is returned.

SetObjectCullMode()

By default, AGK does not attempt to draw the back faces of the polygons that make up a 3D object. Although it is faster to draw only the front of a face, certain models expose the backfaces of some polygons to the camera's view and, in these situations, we need to have the back faces drawn as well.

To set which faces of a polygon are to be drawn, use the `SetObjectCullMode()` statement (see FIG-23.47).

FIG-23.47

SetObjectCullMode()

`SetObjectCullMode (id , imode)`

where

id is an integer value giving the ID of the object.

imode is an integer value (0, 1 or 2) giving the drawing mode to be used (0: draw both front and back faces, 1: draw front faces only, 2: draw back faces only). The default setting is 1.

The program in FIG-23.48 displays a hemisphere open at its top. This opening exposes the back faces of part of the sphere to the camera's view. In order to "see" the inside of the sphere we need to have both the front and back faces of the sphere displayed.

FIG-23.48

Displaying Back Faces

```

rem *** Faces Display Options ***

rem *** Load hemisphere ***
LoadObject(1,"Hemisphere.obj",6)

rem *** Set up light ***
CreateLightDirectional(1,10,-10,20,200,200,200)

rem *** Position camera ***
SetCameraPosition(1,10,10,-20)
SetCameraLookAt(1,0,0,0,0)

rem *** Display front and back faces ***
SetObjectCullMode(1,0)
do
    Sync()
loop

```

Activity 23.19

Start a new project called *Faces3D* and implement the code given above loading *Hemisphere.obj* into the *media* folder.

Test your program observing the appearance of the hemisphere.

Modify the program so that only the front faces are displayed. How does this affect the displayed object?

Finally, change the code so that only back faces are displayed and observe the effect created.

GetObjectCullMode()

To discover the current face-drawing option being applied to a specific object, use `GetObjectCullMode()` (see FIG-23.49).

FIG-23.49`GetObjectCullMode()`

integer `GetObjectCullMode` (`(id)`

where

id

is an integer value giving the ID of the object whose culling mode is to be determined.

The value returned will be 0, 1 or 2.

Transforming Objects

The general term **transform** is used to describe the action performed when an object is moved, rotated or resized. AGK contains several transform statements; these are given below.

SetObjectPosition()

When a 3D object is first created (or loaded) it is centred about the position (0, 0, 0). To centre the object about a new position, use `SetObjectPosition()` (see FIG-23.50).

FIG-23.50

SetObjectPosition()


 A diagram showing the function call `SetObjectPosition (id , x , y , z)`. The function name is in a rounded rectangle. The parameters are enclosed in parentheses and separated by commas. The parameter `id` is in a green box, while `x`, `y`, and `z` are in orange boxes.

where

id is an integer value giving the ID of the object.

x, y, z are real values giving the new position for the object.

Activity 23.20

Modify *Second3D* so that the box appears to be sitting on top of the x-axis (remember the box is 6 units high).

Run and save your program.

MoveObjectLocalX()

While `SetObjectPosition()` places an object at an exact point in 3D space, it is also possible to move an object relative to its current position - in other words, movement is measured from the object's local axes.

Relative movement in the x direction is achieved using `MoveObjectLocalX()` (see FIG-23.51).

FIG-23.51

MoveObjectLocalX()


 A diagram showing the function call `MoveObjectLocalX (id , dist)`. The function name is in a rounded rectangle. The parameters are enclosed in parentheses and separated by a comma. The parameter `id` is in a green box, and `dist` is in an orange box.

where

id is an integer value giving the ID of the object.

dist is a real value giving the distance the object is to be moved in the x direction. Use a negative value to move in the opposite direction.

MoveObjectLocalY()

To move an object a specified distance in the y direction use `MoveObjectLocalY()` (see FIG-23.52).

FIG-23.52

MoveObjectLocalY()


 A diagram showing the function call `MoveObjectLocalY (id , dist)`. The function name is in a rounded rectangle. The parameters are enclosed in parentheses and separated by a comma. The parameter `id` is in a green box, and `dist` is in an orange box.

where

id is an integer value giving the ID of the object.

dist is a real value giving the distance the object is to be moved in the y direction. Use a negative value to move in the opposite direction.

MoveObjectLocalZ()

To move an object a specified distance in the z direction use `MoveObjectLocalZ()` (see FIG-23.53).

FIG-23.53

MoveObjectLocalZ()

where

- id** is an integer value giving the ID of the object.
- dist** is a real value giving the distance the object is to be moved in the z direction. Use a negative value to move in the opposite direction.

Moving an object will always cause that object's local axes to move along with the object itself.

Activity 23.21

Modify *Second3D* creating a function called *HandleBox()* which moves the box 0.1 units by pressing the following keys:

- R (code 82) : 0.1 units to the right along its x-axis (+ve direction)
- L (76) : 0.1 units to the left along its x-axis (-ve direction)
- U (85) : 0.1 units up its y-axis (+ve direction)
- D (68) : 0.1 units down its y-axis (-ve direction)
- I (73) : 0.1 units into the screen along its z-axis (+ve direction)
- O (79) : 0.1 units out of the screen along its z-axis (-ve direction)

Test and save your program.

+ve = positive
-ve = negative

SetObjectRotation()

To rotate an object about its own axes, use `SetObjectRotation()` (see FIG-23.54).

FIG-23.54

SetObjectRotation()

where

- id** is an integer value giving the ID of the object.
- ax** is a real value giving the object's angle of rotation about the x-axis (in degrees).
- ay** is a real value giving the object's angle of rotation about the y-axis.
- az** is a real value giving the object's angle of rotation about the z-axis.

Activity 23.22

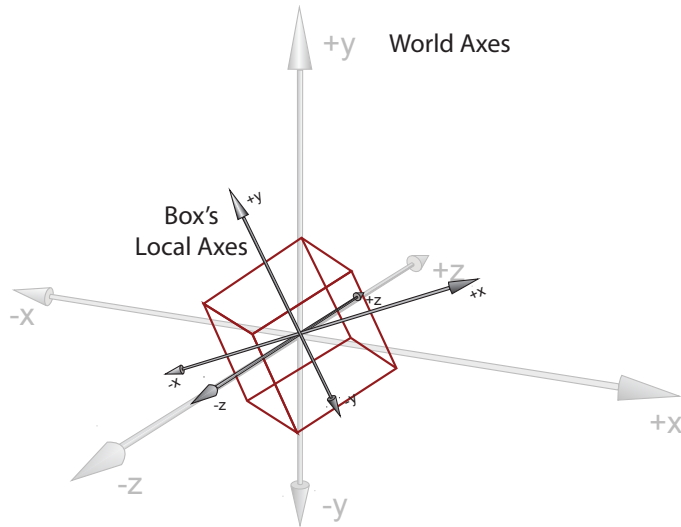
Modify *Second3D* so that the box is initially set to an angle of 20° about its own z-axis. How does this affect the movement of the box when using the control keys set up by Activity 23.21?

Do NOT save this version of your program.

As you can see from the result of Activity 23.22, rotating an object also rotates its local axes. After the 20° rotation, the box's x and y axes are no longer parallel to the world x and y axes (see FIG-23.55).

FIG-23.55

Local Axes Rotate with their Object



With the box in this position, pressing the U key moves the box up its own local y-axis, but since this is no longer parallel to the world axes, we see the box move towards the top and left of the screen. A similar effect is created when we move the box left and right, with it following the new orientation of its local x-axis. Only movement along the z-axis is unchanged since it was about this axis the original rotation was defined.

When the `SetObjectRotation()` statement is used on an object, the object's axes are always assumed to start parallel to the world axes when calculating the new position of the object; any previous rotations created using this command have no effect on the result.

When implementing the rotation, the object is first rotated about its y-axis, then its x-axis and finally its z-axis. We can see the stages involved in executing the statement

```
SetObjectRotation(2,60,20,45)
```

in FIG-23.56.

FIG-23.56

How an Object is Rotated Using `SetObjectRotation()`

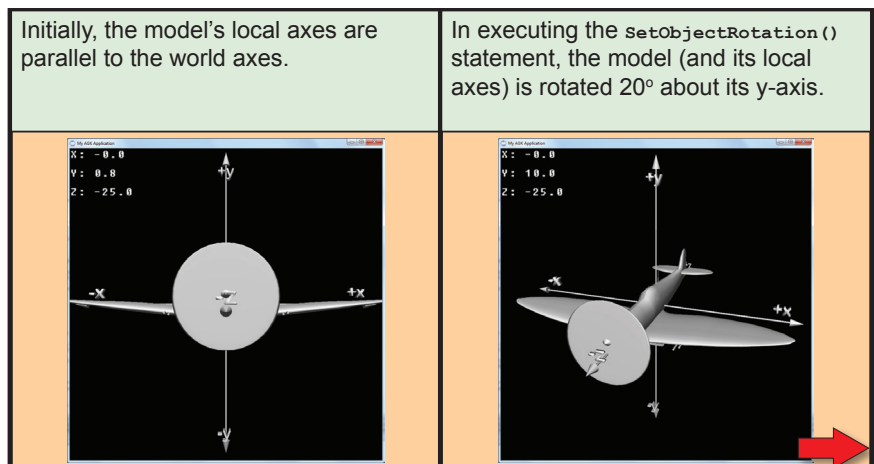
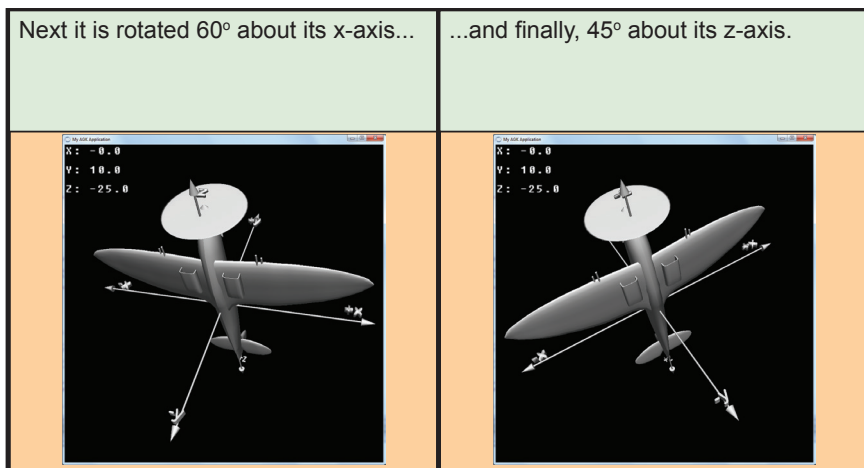


FIG-23.56

(continued)

How an Object
is Rotated Using

`SetObjectRotation()`



RotateObjectLocalX()

As well as rotating an object to a specific orientation, you can also rotate it relative to its current position. To rotate an object further about its local x-axis, use `RotateObjectLocalX()` (see FIG-23.57).

FIG-23.57

`RotateObjectLocalX()`

`RotateObjectLocalX` (`id` , `ang`)

where

id is an integer value giving the ID of the object.

ang is a real value giving the angle through which the object is to be rotated about its x-axis. Use a negative value to rotate in the opposite direction.

RotateObjectLocalY()

To rotate an object further about its y-axis, use `RotateObjectLocalY()` (see FIG-23.58).

FIG-23.58

`RotateObjectLocalY()`

`RotateObjectLocalY` (`id` , `ang`)

where

id is an integer value giving the ID of the object.

ang is a real value giving the angle through which the object is to be rotated about its y-axis. Use a negative value to rotate in the opposite direction.

RotateObjectLocalZ()

To rotate an object further about the z-axis, use `RotateObjectLocalZ()` (see FIG-23.59).

FIG-23.59

`RotateObjectLocalZ()`

`RotateObjectLocalZ` (`id` , `ang`)

where

id is an integer value giving the ID of the object.

ang is a real value giving the angle through which the object is to be rotated about its z-axis. Use a negative value to rotate in the opposite direction.

Activity 23.23

Modify *Second3D* so that the box can be rotated 1° about its y-axis using the Q (code 81)($+1^\circ$) and W (87)(-1°) keys; 1° about its x-axis using E (69)($+1^\circ$) and C (67)(-1°); and 1° about its z-axis using K (75)($+1^\circ$) and M (77)(-1°).

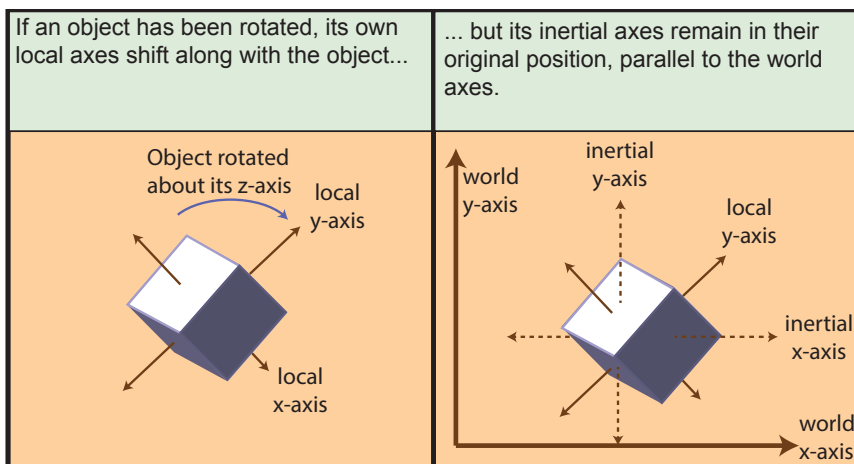
Test and save your program.

RotateObjectGlobalX()

All the rotation statements covered so far rotate an object about its own local axes. Now imagine every object has its own second set of axes which are not affected by that object's rotation. These axes are sometimes known as **inertial axes** (see FIG-23.60).

FIG-23.60

Inertial Axes



The next three statements allow you to rotate an object about its inertial axes.

To rotate an object by a specified angle about its inertial x-axis, use `RotateObjectGlobalX()`. The statement's format is given in FIG-23.61.

FIG-23.61

`RotateObjectGlobalX()`

where

id is an integer value giving the ID of the object.

ang is a real value giving the angle (in degrees) through which the object is to be rotated about its inertial x-axis. Use a negative value to rotate in the opposite direction.

RotateObjectGlobalY()

To rotate an object about its inertial y-axis, use `RotateObjectGlobalY()` (see FIG-23.62).

FIG-23.62

`RotateObjectGlobalY()`

`RotateObjectGlobalY` (`id` , `ang`)

where

id	is an integer value giving the ID of the object.
ang	is a real value giving the angle through which the object is to be rotated about its inertial y-axis. Use a negative value to rotate in the opposite direction.

RotateObjectGlobalZ()

To rotate an object about its inertial z-axis, use `RotateObjectGlobalZ()` (see FIG-23.63).

FIG-23.63

`RotateObjectGlobalZ()`

`RotateObjectGlobalZ` (`id` , `ang`)

where

id	is an integer value giving the ID of the object.
ang	is a real value giving the angle through which the object is to be rotated about its inertial z-axis. Use a negative value to rotate in the opposite direction.

Activity 23.24

Modify *Second3D* so that the box can be:

- rotated 1° about its inertial x-axis using Z (90)(+1°) and X (88)(-1°) keys;
- 1° about its inertial y-axis using F (70)(+1°) and V (86)(-1°);
- 1° about its inertial z-axis using B (66)(+1°) and N (78)(-1°).

Test and save your program.

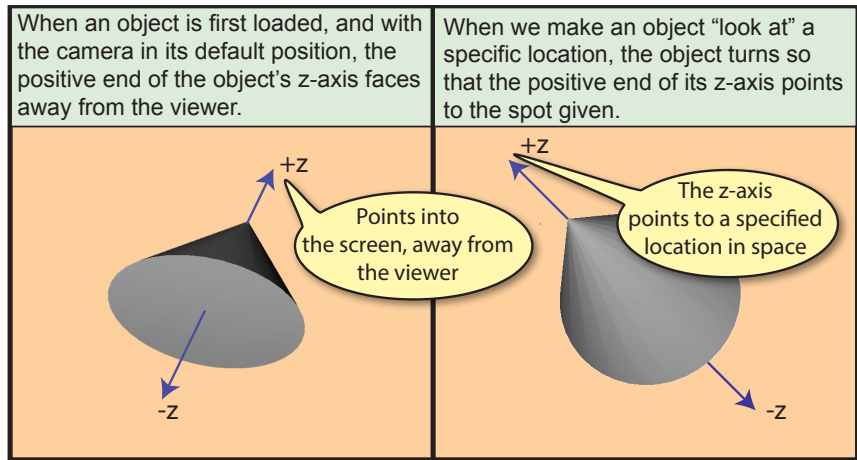
SetObjectLookAt()

There are occasions when we want an object to face in a particular direction. For example, imagine that we want an enemy robot always to face towards your character when playing a shooting game. Obviously moving the model robot would involve rotating it appropriately but it would be difficult to calculate exactly what angles of rotation were required.

To overcome this problem, the `SetObjectLookAt()` allows us to have an object face a specific point in space. Using this command causes an object to turn in such a way as to ensure that the positive end of its local z-axis is aimed at a specific point in space. In effect, this means that the side of the object that is initially facing away from the camera turns to face a specific location (see FIG-23.64).

FIG-23.64

Making an Object "Look At" a Point in Space



In addition, `SetObjectLookAt()` allows the model to be rotated to a specific angle about its own z-axis. The statement has the format shown in FIG-23.65.

FIG-23.65

`SetObjectLookAt()`

`SetObjectLookAt (id , x , y , z , roll)`

where

- id** is an integer value giving the ID of the object.
- x, y, z** are real values giving the coordinates of the point to which the object is to face.
- roll** is a real value giving the angle of roll for the object.

We'll look at a program example using this statement shortly.

GetObjectX(), GetObjectY() and GetObjectZ()

To find the location of a 3D object use the three statements `GetObjectX()`, `GetObjectY()` and `GetObjectZ()`.

The format for these three statements are given in FIG-23.66.

FIG-23.66

`GetObjectX()`
`GetObjectY()`
`GetObjectZ()`

float `GetObjectX (id)`
float `GetObjectY (id)`
float `GetObjectZ (id)`

where

- id** is an integer value giving the ID of the 3D object whose position is to be found.

Each statement returns one of the coordinates of the specified 3D object.

The program in FIG-23.67 has a model in the shape of a box with a projecting cone “looking at” a moving sphere.

FIG-23.67

Using the “Look At”
Option

```
rem *** Object Look At ***

rem *** Create object that looks ***
LoadObject(1,"SpikedBox.obj",6)
SetObjectPosition(1,0,0,20)

rem *** Create object looked at ***
CreateObjectSphere(2,1,10,10)
SetObjectPosition(2,-10,3,-15)

rem *** Position camera ***
SetCameraPosition(1,0,10,-40)
SetCameraLookAt(1,0,0,0,0)

rem *** Set up light ***
CreateLightDirectional(1,10,-10,20,200,200,200)

rem *** x coord for sphere ***
x# = -10

rem *** Move sphere and have spiked box look at it ***
do
    rem *** Move sphere ***
    SetObjectPosition(2,x#,3,-15)
    rem *** Make spiked box look at sphere ***
    SetObjectLookAt(1,GetObjectX(2), GetObjectY(2),-15,0)
    rem *** Update screen ***
    Sync()
    rem *** wait 50 msec ***
    sleep(50)
    rem *** Increment x coord ***
    x#=x#+0.125
loop
```

Activity 23.25

Start a new project, *Follow3D*, and implement the code given in FIG-23.67 (copy *SpikedBox.obj* into the project’s *media* folder).

Test and save your program.

GetObjectAngleX(), GetObjectAngleY() and GetObjectAngleZ()

If you want to discover the angles that an object has been rotated around each of its own local axes, use the `GetObjectAngleX()`, `GetObjectAngleY()` and `GetObjectAngleZ()` statements (see FIG-23.68).

FIG-23.68

GetObjectAngleX()
GetObjectAngleY()
GetObjectAngleZ()

float `GetObjectAngleX()` (id)

float `GetObjectAngleY()` (id)

float `GetObjectAngleZ()` (id)

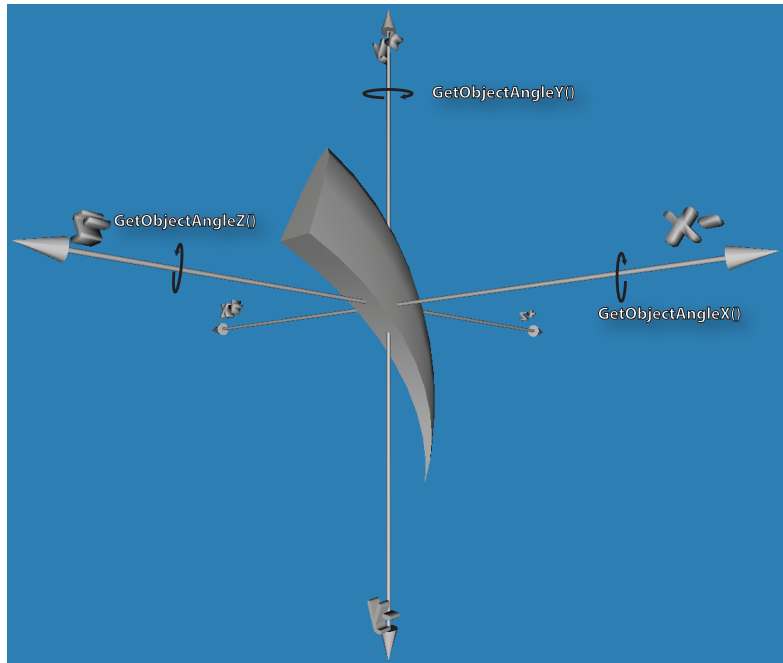
where

id is an integer value giving the ID of the 3D object whose rotations are to be found.

The values retrieved by these commands are visualised in FIG-23.69.

FIG-23.69

The Values Returned by the `GetObjectAngle` statements.



SetObjectScale()

To resize an object after it has been created, use `SetObjectScale()` (see FIG-23.70).

FIG-23.70

`SetObjectScale()`

`SetObjectScale ((id , sx , sy , sz)`

where

id is an integer value giving the ID of the 3D object to be scaled.

sx is a real number giving the scaling to be performed in the x direction (object width).

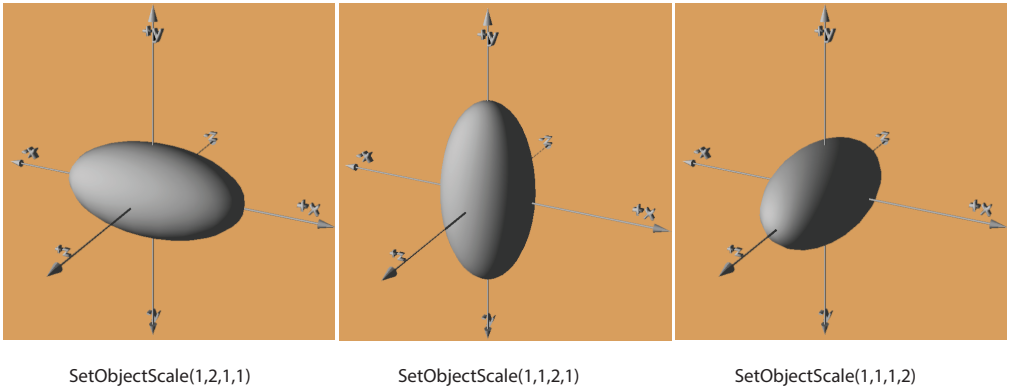
sy is a real number giving the scaling to be performed in the y direction (object height).

sz is a real number giving the scaling to be performed in the z direction (object depth).

Scaling factors are based on the original size of the 3D object when it was first loaded or created. A scaling value of 1 sets an object to its original size in the specified direction; a value of 2 would double the object's size in that direction; a value of 0.5 would halve it.

FIG-23.71 shows the effect on a sphere of doubling each dimension in turn.

FIG-23.71 Effects of Scaling



The program in FIG-23.72 reshapes a sphere over time, first stretching and then shrinking it in the x direction.

FIG-23.72

Using Object Scaling

```
rem *** Object Scaling ***

rem *** Load axes ***
LoadObject(2,"Axes.obj",15)

rem *** Create a sphere ***
CreateObjectSphere(1,4,20,20)

rem *** Set up light ***
CreateLightDirectional(1,10,-10,20,200,200,200)
SetObjectScale(1,1,1,2)

rem *** Position camera ***
SetCameraPosition(1,10,10,-20)
SetCameraLookAt(1,0,0,0,0)

rem *** Set object scaling factors ***
scale# = 0.1
change# = 0.01

rem *** Scale object over time ***
do
  SetObjectScale(1,scale#,1,1)
  Sync()
  Sleep(10)
  scale# = scale# + change#
  if scale# > 2
    change# = -0.01
  elseif scale# < 0.1
    change# = 0.01
  endif
loop
```

Activity 23.26

Start a new project, *Scale3D*, and implement the code given in FIG-23.72.

Modify the code so that expansion happens in the z direction instead of the x direction. Test and save your program.

- Use `CreateObjectBox()` to create a box object.
- Use `CreateObjectSphere()` to create a sphere.
- The number of rows and columns specified when creating a sphere will affect the accuracy of the sphere's shape.
- Use `CreateObjectCone()` to create a cone object.
- The number of segments specified when creating a cone will affect the accuracy of the cone's shape.
- Use `CreateObjectCylinder()` to create a cylinder object.
- The number of segments specified when creating a cylinder will affect the accuracy of the cylinder's shape.
- Use `CreateObjectPlane()` to create a plane object.
- Planes are created in a vertical position (in the XY plane).
- Use `CloneObject()` to create an independent copy of an existing object.
- Use `InstanceObject()` to create a copy of an existing object which retains links to the original object.
- Deleting an object from which an instanced object has been created will cause the instanced object also to be deleted.
- Use `GetObjectExists()` to check if an object of a specified ID currently exists.
- Use `DeleteObject()` to delete an existing object.
- Use `SetObjectColor()` to specify a colour for the surface of an object.
- Use `SetObjectImage()` to texture an object with an existing image.
- When using an image to texture an object, set the stage parameter to 0.
- Other images can be assigned to stages 1 to 7 of an object. These can then be used by a shader to affect the appearance of the object.
- Use `SetObjectTransparency()` to activate the transparency assigned to an object from the `SetObjectColor()` or `SetObjectImage()` statements.
- Use `GetObjectTransparency()` to determine the current transparency setting of an object.
- Use `SetObjectVisible()` to make an object visible/invisible.
- Use `GetObjectVisible()` to determine if an object is currently visible.
- Use `SetObjectCullMode()` to determine which faces of an object are to be displayed.
- Normally, objects only need their front faces to be displayed, but some may need back faces to be displayed as well.
- Use `GetObjectCullMode()` to determine which faces of an object are currently being drawn.
- Use `SetObjectPosition()` to reposition an existing object.

- Objects default to being positioned with their centres at the origin.
- Use `MoveObjectLocalX()`, `MoveObjectLocalY()` and `MoveObjectLocalZ()` to move an object relative to its own local axes.
- Use `SetObjectRotation()` to specify the angle an object is to be rotated about its local axes with angles being measured from the axes' original orientation parallel to the world axes.
- When an object is rotated using `SetObjectRotation()`, it is first rotated about its y-axis, then its x-axis and finally, its z-axis.
- To rotate an object relative to its current rotation, use `RotateObjectLocalX()`, `RotateObjectLocalY()` and `RotateObjectLocalZ()`.
- An object's inertial axes are similar to its local axes in that they have their origin at the centre of the object, but inertial axes are unaffected by the object's rotation and remain parallel to the world axes.
- To rotate an object relative to its inertial axes, Use `RotateObjectGlobalX()`, `RotateObjectGlobalY()` and `RotateObjectGlobalZ()`.
- Use `SetObjectLookAt()` to make the positive end of an object's z-axis aim at a specific point in space.
- To find the location of an object in space, use `GetObjectX()`, `GetObjectY()` and `GetObjectZ()`.
- To find out the angle to which an object has been rotated about each of its own axes, use `GetObjectAngleX()`, `GetObjectAngleY()` and `GetObjectAngleZ()`.

Cameras

Introduction

We already know that it is the virtual camera that creates the image we see on the screen and that it can be moved and made to point at a specific position in space. However, several more camera-related commands exist and these are detailed below.

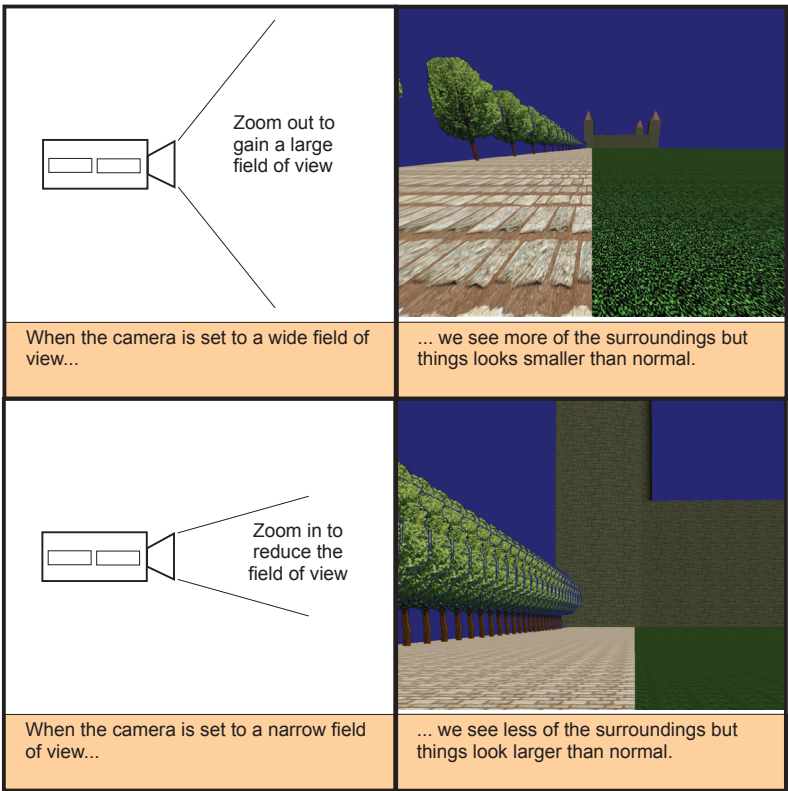
Camera-Related Statements

SetCameraFOV()

In the real world, most cameras come equipped with a zoom lens which allows the user to zoom in on distant objects or zoom out to take in all the features within a restricted area such as a room. Exactly what is captured by the camera is determined by the camera's field of view (FOV) and this is measured as the angle between the left and right edges of the image (see FIG-23.73).

FIG-23.73

Using Zoom



You can make a camera zoom in or out by changing its field of view. This is done using the `SetCameraFOV()` statement (see FIG-23.74).

FIG-23.74

`SetCameraFOV()`

`SetCameraFOV (id , fov)`

where

id is an integer value giving the ID of the camera.

fov

is a real number giving the viewing angle of the camera. The default value is 45°; a smaller number will make the camera zoom in.

The program in FIG-23.75 demonstrates the effect of changing the field of view. Initially it starts with a setting of 90° and gradually reduces to 10°.

FIG-23.75

Zooming

```
rem *** FOV Demo ***

rem *** Load Model ***
LoadObject(1,"Robot.obj",15)
rem *** Apply texture ***
LoadImage(1,"Robotskin.png")
SetObjectImage(1,1,0)
rem *** Create textured background plane ***
CreateObjectPlane (2,100,100)
SetObjectPosition(2,0,0,10)
LoadImage(2,"Background.jpg")
SetObjectImage(2,2,0)

rem *** Create Directional light ***
CreateLightDirectional(1,10,10,10,255,255,255)

rem *** Position and point camera ***
SetCameraPosition(1,0,10,-30)
SetCameraLookAt(1,0,5,0,0)

rem *** Set field of view ***
fov = 90
rem *** Zoom in ***
do
    SetCameraFOV(1,fov)
    Sync()
    Sleep(50)
    if fov > 10
        dec fov
    endif
loop
```

Activity 23.27

Start a new project, *Zoom3D*, and implement the code given in FIG-23.75 copying the required files to the *media* folder.

Modify *Zoom3D* so a function called *HandleZoom()* allows the zoom factor to be controlled from the keyboard. Zoom in (decreasing angle) using the *PageUp* key (code 33) and zoom out using the *Page Down* key (34). The zoom should be incremented in 1° steps and constrained to be in the range 5 to 100.

Test and save your program.

SetCameraRange()

You can make the camera ignore items which are very close or very far away. In effect, anything that lies outside the specified range is invisible to the camera and will not appear on the screen. The space between the closest and furthest points is known as the **range** of the camera. To change the range of a camera use `SetCameraRange()` (see FIG-23.76).

FIG-23.76

SetCameraRange()

SetCameraRange ((id , near , far))

where

- id** is an integer value giving the ID of the camera.
- near** is a real number giving the nearest point to the camera that is to be visible.
- far** is a real number giving the farthest point that is to be visible.

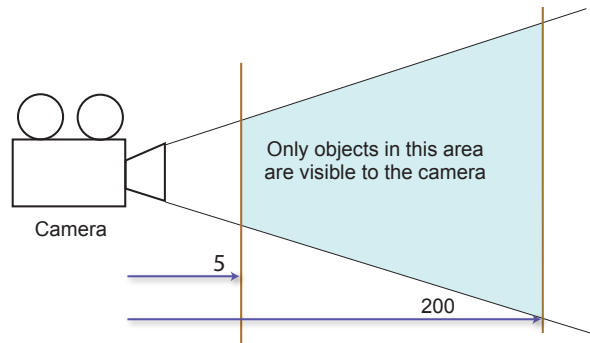
For example, the statement

```
SetCameraRange(1,5,200)
```

would create the effect shown in FIG-23.77.

FIG-23.77

The Effects of Using SetCameraRange()



If only part of an object falls within the specified range, then only that part of the object will be displayed.

The program in FIG-23.79 creates a cone, box and sphere at varying distances along the z-axis. The user can then toggle between changing the camera's near or far range by pressing the *Enter* key. The range setting for the selected limit can then be incremented or decremented by pressing the + or - keys respectively.

The results produced by various settings are shown in FIG-23.78.

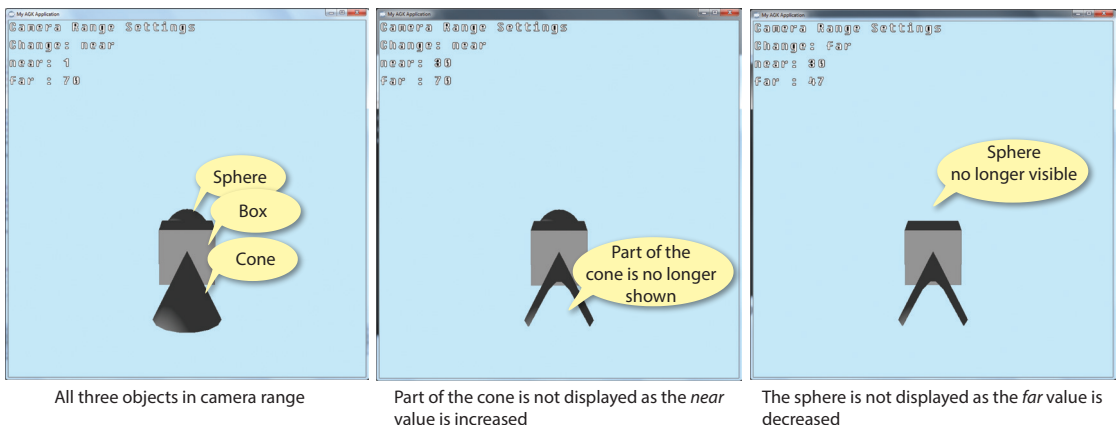
FIG-23.78 Effect of Range Changes

FIG-23.79

Adjusting the Camera's
Range

```

rem *** Demonstrating Camera Range ***

rem *** Camera Range Data Structure ***
type CameraRangeType
    near as float
    far as float
endtype

rem *** Global variables ***
rem *** Camera Range Info ***
global range as CameraRangeType
range.near = 1
range.far = 70
rem *** Range being changed ***
global limittochange = 1 // 1:near; 2:far

rem *** Range Text used in display ***
dim changetext[3] as string = ["", "near", "far"]
changetext[1]= "near"
changetext[2] = "far"

rem *** Create box at origin ***
CreateObjectBox(1,5,5,5)

rem *** Create Cone at -10 on z-axis ***
CreateObjectCone(2,5,5,15)
SetObjectPosition(2,0,0,-10)

rem *** Create sphere at +10 on z-axis ***
CreateObjectSphere(3,5,12,12)
SetObjectPosition(3,0,0,10)

rem *** Create Directional light ***
CreateLightDirectional(1,10,10,10,255,255,255)

rem *** Position and point camera ***
SetCameraPosition(1,0,10,-40)
SetCameraLookAt(1,0,5,0,0)

rem *** Create text objects to display range ***
CreateText(1,"Camera Range Settings")
CreateText(2,"Change: near")
SetTextPosition(2,0,5)
CreateText(3,"near:")
SetTextPosition(3,0,10)
CreateText(4,"far:")
SetTextPosition(4,0,15)

rem *** Set initial camera range ***
SetCameraRange(1,range.near,range.far)

do
    HandleCameraRange()
    rem *** Update range strings ***
    SetTextString(2,"Change: "+changetext[limittochange])
    SetTextString(3,"near: "+str(range.near))
    SetTextString(4,"far : "+str(range.far))
    rem *** Update screen ***
    Sync()
loop

```

FIG-23.79

(continued)

Adjusting the Camera's Range

```

function HandleCameraRange()
    rem *** Get key pressed ***
    key = GetRawLastKey()
    select key
        case 13: //return key, change selected
            rem *** If key just been pressed ***
            if GetRawKeyPressed(key) = 1
                rem *** Change limit selected ***
                limittochange = 3 - limittochange
            endif
        endcase
        case 187: // + Key, increment
            rem *** If key pressed ***
            if GetRawKeyState(key) = 1
                rem *** increment selected limit ***
                if limittochange = 1
                    inc range.near
                else
                    inc range.far
                endif
                Sleep(50)
            endif
        endcase
        case 189: // - key, decrement
            rem *** If key pressed ***
            if GetRawKeyState(key) = 1
                rem *** decrement selected limit ***
                if limittochange = 1
                    dec range.near
                else
                    dec range.far
                endif
            endif
            sleep(50)
        endcase
    endselect
    rem *** Adjust camera range to match change ***
    SetCameraRange(1, range.near, range.far)
endfunction

```

Activity 23.28

Start a new project, *Range3D*, and implement the code given in FIG-23.79. Run the program and observe the results of changing the range values.

Modify the program so that both front and back faces are displayed. Test and save your program.

SetCameraRotation()

To rotate a camera to a specific angle about its local axes, use `SetCameraRotation()` (see FIG-23.80).

FIG-23.80

SetCameraRotation()

`SetCameraRotation` (`(` `id` `,` `ax` `,` `ay` `,` `az` `)`

where

id

is an integer value giving the ID of the camera (the default camera has an ID of 1).

ax	is a real value giving the camera's angle of rotation about its x-axis (in degrees).
ay	is a real value giving the camera's angle of rotation about its y-axis.
az	is a real value giving the camera's angle of rotation about its z-axis.

RotateCameraLocalX(), RotateCameraLocalY() and RotateCameraLocalZ()

If you want to rotate a camera about its local axes relative to its current angle of rotation, then you can use `RotateCameraLocalX()`, `RotateCameraLocalY()`, or `RotateCameraLocalZ()` as appropriate (see FIG-23.81).

FIG-23.81

`RotateCameraLocalX()`
`RotateCameraLocalY()`
`RotateCameraLocalZ()`

```
RotateCameraLocalX ( ( id , ang ) )
RotateCameraLocalY ( ( id , ang ) )
RotateCameraLocalZ ( ( id , ang ) )
```

where

id	is an integer value giving the ID of the camera (the default camera has an ID of 1).
ang	is a real value giving the degrees by which the camera is to be rotated. The angle is measured from the current angle of rotation. The axis about which rotation takes places depends on the statement selected.

For example, the statement

```
RotateCameraLocalY(1,10)
```

would rotate the camera a further 10° about its own y-axis.

RotateCameraGlobalX(), RotateCameraGlobalY() and RotateCameraGlobalZ()

If you need the camera to rotate about its inertial axes (which always remain parallel to the world axes), then use `RotateCameraGlobalX()`, `RotateCameraGlobalY()` or `RotateCameraGlobalZ()`. The format for each of these statements is given in FIG-23.82.

FIG-23.82

`RotateCameraGlobalX()`
`RotateCameraGlobalY()`
`RotateCameraGlobalZ()`

```
RotateCameraGlobalX ( ( id , ang ) )
RotateCameraGlobalY ( ( id , ang ) )
RotateCameraGlobalZ ( ( id , ang ) )
```

where

id	is an integer value giving the ID of the camera.
ang	is a real value giving the angle through which the camera is to be rotated. The axis about which rotation takes places depends on the statement selected.

MoveCameraLocalX(), MoveCameraLocalY() and MoveCameraLocalZ()

While `SetCameraPosition()` moves a camera to a specified point in space, if you want to move a camera relative to its current position, use the statements `MoveCameraLocalX()`, `MoveCameraLocalY()` or `MoveCameraLocalZ()` as appropriate.

The format of each of these statements is shown in FIG-23.83).

FIG-23.83

`MoveCameraLocalX()`
`MoveCameraLocalY()`
`MoveCameraLocalZ()`

The diagram illustrates the syntax for three statements: `MoveCameraLocalX()`, `MoveCameraLocalY()`, and `MoveCameraLocalZ()`. Each statement is shown in a rounded rectangular box, followed by an opening parenthesis '(', then a box labeled 'id', a comma ',', a box labeled 'dist', and a closing parenthesis ')'. The 'id' and 'dist' boxes are highlighted in green.

where

id	is an integer value giving the ID of the camera.
dist	is a real value giving the distance which the camera is to be moved. The direction of the movement is determined by which of the three statements is selected.

GetCameraX(), GetCameraY() and GetCameraZ()

To retrieve the camera's position in space, use the statements `GetCameraX()`, `GetCameraY()` and `GetCameraZ()` (see FIG-23.84).

FIG-23.84

`GetCameraX()`
`GetCameraY()`
`GetCameraZ()`

The diagram illustrates the syntax for three statements: `GetCameraX()`, `GetCameraY()`, and `GetCameraZ()`. Each statement is preceded by the keyword 'float' and followed by a rounded rectangular box containing the statement name, an opening parenthesis '(', a box labeled 'id', and a closing parenthesis ')'. The 'id' boxes are highlighted in green.

where

id	is an integer value giving the ID of the camera.
-----------	--

GetCameraAngleX(), GetCameraAngleY(), and GetCameraAngleZ()

To discover the angle that a camera has been rotated around each of its own local axes, use the `GetCameraAngleX()`, `GetCameraAngleY()` and `GetCameraAngleZ()` statements (see FIG-23.85).

FIG-23.85

GetCameraAngleX()
GetCameraAngleY()
GetCameraAngleZ()

float **GetCameraAngleX** ((id))
float **GetCameraAngleY** ((id))
float **GetCameraAngleZ** ((id))

where

id

is an integer value giving the ID of the camera whose rotations are to be found.

GetObjectInScreen()

You can check if an object is within the camera's view (and hence, appears on the screen), using the **GetObjectInScreen** () statement (see FIG-23.86).

FIG-23.86

GetObjectInScreen()

integer **GetObjectInScreen** ((id))

where

id

is an integer value giving the ID of the object to be checked.

The statement returns 1 if the object is on screen, 0 if it is not.

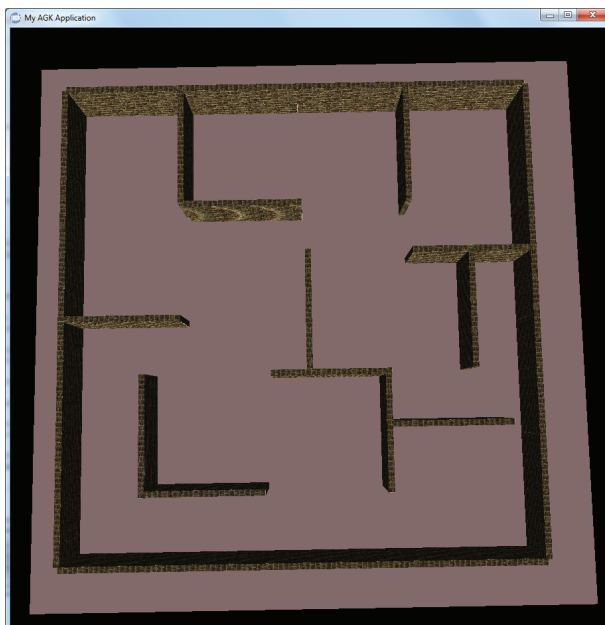
Using Camera Commands to Create First Person Perspective

Initial Set Up

The camera commands can be used to allow us to create a first-person perspective game. In the next few pages we will develop the basic movement strategies for such a game, allowing the user to manoeuvre the camera within the environment shown in FIG-23.87.

FIG-23.87

Maze Layout



Instead of simply presenting the final code, we will, instead, build incrementally towards the final result. The code in FIG-23.88 creates the layout and positions the camera within the model.

FIG-23.88

Setting Up the Maze

```
rem *** First Person Perspective ***

rem *** Load textured walls ***
LoadObject(1,"Maze.obj",20)
LoadImage(1,"BricksLarge.jpg")
SetObjectImage(1,1,0)
rem *** Load floor ***
CreateObjectPlane(2,180,180)
RotateObjectLocalX(2,90)
SetObjectPosition(2,0,1,0)
SetObjectColor(2,220,180,180,0)

rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,250,250,250)

rem *** Position camera ***
SetCameraPosition(1,5,5,-10)
SetCameraLookAt(1,5,5,-10,0)

rem *** View scene ***
do
    Sync()
loop
```

Activity 23.29

Start a new project, *FP3D*, and implement the code given in FIG-23.88.

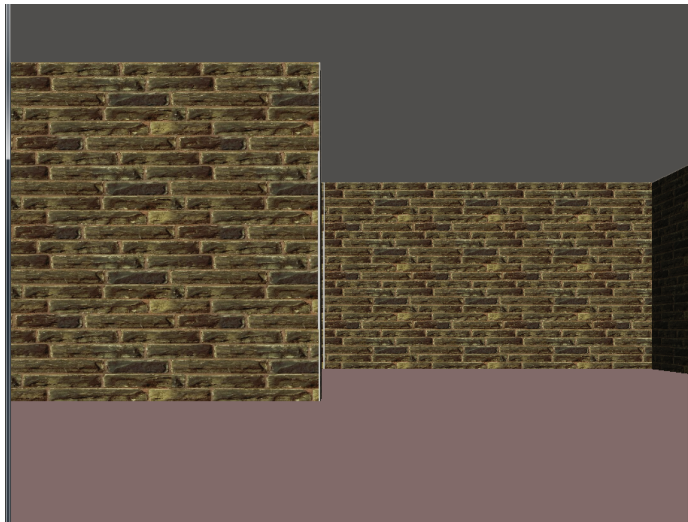
Compile the code then copy the files *Maze.obj* and *BricksLarge.jpg* into the project's *media* folder.

Run and save your code.

The starting view when the code is run is shown in FIG-23.89.

FIG-23.89

Player's View



Setting the Field of View

A human being has a field of view of about 180° but the left and right edges of this area can be seen by one eye only. If we consider the area which can be seen by both eyes simultaneously, then our field of view is about 120°.

With the virtual camera defaulting to a much narrower field of view, we need to begin by increasing this figure, although you may feel setting it to 120° is too much for a game environment since everything seems much further away. Also, in a shooting game, a wide field of view can make aiming more difficult. A good compromise would be between 70° and 90°, so we will settle for the half way value and set the field of view to 80° using the statements:

```
rem *** Set camera's field of view to 80 ***
SetCameraFOV(1,80)
```

Activity 23.30

Modify *FP3D* so that the camera's field of view is set to 80°.

Test your program, checking how this changes what is visible to the player.

Save your project.

Turning the Camera

To allow the player to look around his environment, we need to be able to rotate the camera about its y-axis.

We have a choice of rotating the camera using either the `RotateCameraLocalY()` to rotate the camera about its local y-axis or `RotateCameraGlobalY()` to rotate it about its inertial y-axis. At this point, both options will create the same effect, since both the local and inertial axes are parallel to the world axes. However, as we shall see later, the local y-axis may shift orientation making `RotateCameraGlobalY()` the best option when turning the camera.

The rotation can be initiated by using the left and right arrow keys and as in previous programs, we will make use of a separate function:

```
function HandleCamera()
  rem *** Get last key pressed ***
  key = GetRawLastKey()
  rem *** If key currently pressed, process it***
  if GetRawKeyState(key)=1
    select key
      rem *** Turn camera ***
      case 37: //left cursor, turn camera left
        RotateCameraGlobalY(1,-1)
      endcase
      case 39: //right cursor, turn camera right
        RotateCameraGlobalY(1,1)
      endcase
    endselect
  endif
endfunction
```

which will be called from within the main section's `do...loop` structure.

Activity 23.31

Modify *FP3D* using the code given above to allow the player to look around the playing area.

Test and save your program.

Moving

Next, the user needs to be able to move about within his environment. To do this we need to allow the camera to be moved forward or backward along the direction in which it is looking - in other words, along its local z-axis. This requires the following `case` options to be added to the `select` statement:

```
rem *** Move camera options ***
case 87: //W key, Camera forward ***
    MoveCameraLocalZ(1,0.2)
endcase
case 83: //S key, Camera back ***
    MoveCameraLocalZ(1,-0.2)
endcase
```

Note that the camera is moved 0.2 units per frame.

Activity 23.32

Modify *FP3D* so that the camera can be moved forward and backward.

Can the camera pass through the walls?

Test and save your program.

We will return to the problem of being able to pass ghost-like through the walls later.

Looking Up and Down

Another common option is to allow the player to look up or down. Since this simulates the tilting of a human head by rotating the camera about its own x-axis, we need to limit the degree of rotation allowed to no more than 45° up or down. In a game environment even less of a rotation is required.

We'll use the up and down cursor keys for this operation, limiting the angle to $\pm 30^\circ$. The operation causes the camera to rotate about its own x-axis and uses the following code:

```
case 38: //up cursor,tilt camera up
    if GetCameraAngleX(1) > -30
        RotateCameraLocalX(1,-1)
    endif
endcase
case 40: //down cursor,tilt camera down
    if GetCameraAngleX(1) < 30
        RotateCameraLocalX(1,1)
    endif
endcase
```


Note that a negative value is required to tilt the camera upward.

Activity 23.33

Modify *FP3D* so that the camera can look up and down.

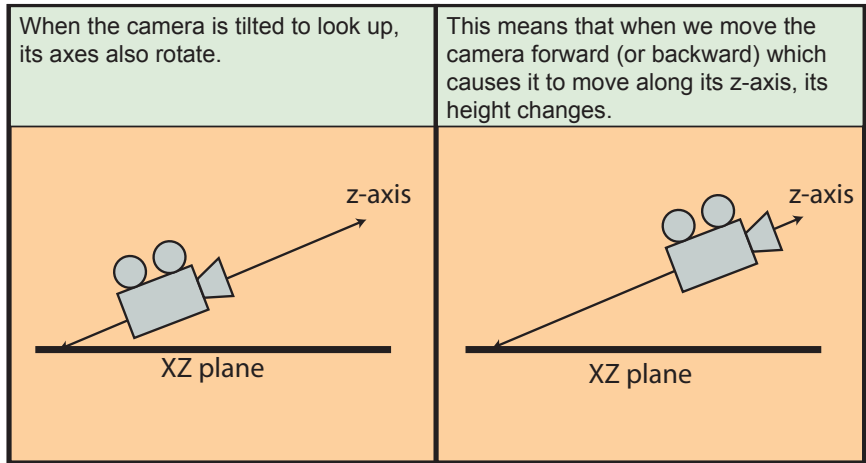
What happens when the camera is pointed downward and then made to travel forward?

Test and save your program.

As you can see from the results of Activity 23.33, moving forward (or backward) when the camera is not pointing straight forward (0° rotation about its x-axis) the height of the camera changes as it is moved. The reason for this is shown in FIG-23.90.

FIG-23.90

How the Camera Moves when Tilted



We avoided the same type of problem when turning the camera by rotating the camera about its inertial y-axis rather than its local y-axis. However, we cannot use the inertial z-axis for the camera movement, since it will not be pointing in the appropriate direction once the player uses the left and right cursor keys to turn the camera.

To stop a tilted camera changing height as it is moved, we must first make sure it is pointing straight ahead. In other words, we must undo any up or down tilt operation. This will ensure that the local z-axis is parallel to the floor of the maze and that the height of the camera will not change as it moves. We can then move the camera before finally restoring the original tilt to the camera.

These new requirements require the *W* (forward) and *S* (backward) keys to be handled by the following modified code:

```
case 87: //W key, Camera forward ***
    rem *** Make camera parallel to floor ***
    angle = GetCameraAngleX(1)
    RotateCameraLocalX(1,-angle)
    rem *** Move camera ***
    MoveCameraLocalZ(1,0.2)
    rem *** Camera to original tilt ***
    RotateCameraLocalX(1,angle)
endcase
case 83: //S key, Camera back ***
    rem *** Make camera parallel to floor ***
```

```

angle = GetCameraAngleX(1)
RotateCameraLocalX(1,-angle)

rem *** Move camera ***
MoveCameraLocalZ(1,-0.2)
rem *** Camera to original tilt ***
RotateCameraLocalX(1,angle)
endcase

```

Activity 23.34

Modify *FP3D*'s *HandleCamera()* function so that the camera moves correctly when tilted up or down.

Test and save your program.

Billboarding

Sometimes objects in a 3D game are not all they appear to be. Elements that look 3D are in fact 2D images textured onto a plane. For example, the screen shot in FIG-23.91 shows an image of a tree textured onto a plane.

FIG-23.91

A Tree Billboard



Activity 23.35

Modify *FP3D* by creating a 10 x 20 plane (id = 3), positioning it at (10,11,0).

Texture the plane using the image *Tree.png* (copying the file to the *media* folder). Run the program and move around the tree. Why is the tree not a realistic model?

Save your program.

Of course, we can see that the illusion of a tree doesn't work because as we move about we become aware of the two-dimensional properties of the plane.

However, we might be able to overcome this problem by ensuring that the plane remains looking at the camera. We can do this by adding the lines:

```
rem *** Turn tree to look at camera ***
SetObjectLookAt(3,GetCameraX(1),5,GetCameraZ(1),0)
```

Activity 23.36

Modify *FP3D* by adding a new function called *HandleTree()* which contains the two statements given above. Call the function at the end of the `if GetRawKeyState(key)=1` structure within *HandleCamera()*.

Does this solve the problem?

Save your program.

This time our problem is that the tree will tilt about its x and y axes to ensure the plane is perpendicular to the specified point it is looking at. To solve this problem we need to ensure that it only revolves about its y-axis and remains perpendicular to the ground.

To do this we only need to add the lines

```
rem *** Undo any rotation about the x-axis ***
SetObjectRotation(3,0,GetObjectAngleY(3),0)
```

after making the tree look at the camera.

Activity 23.37

Modify *FP3D*'s *HandleTree()* by adding the two statements given above at the end of the function.

Does this solve the problem?

Save your program.

This technique of using a 2D object to represent a 3D shape is known as **billboarding** and works best when the player cannot get too close to the billboard object.

Summary

- The view displayed on the screen represents the part of the 3D world captured by the virtual camera.
- The current version of AGK (v108) uses a single virtual camera but later versions will offer more.
- Use `SetCameraFOV()` to set the camera's field of view (this allows you to zoom in or out).
- Use `SetCameraRange()` to set the near and far distances between which objects are visible to the camera.

- Use `SetCameraRotation()` to set the camera's rotation to a absolute angle (given in degrees) about each of its local axes.
- Use `RotateCameraLocalX()`, `RotateCameraLocalY()`, and `RotateCameraLocalZ()` to change the camera's rotation about a specific local axis by a specified amount (given in degrees).
- Use `RotateCameraGlobalX()`, `RotateCameraGlobalY()`, and `RotateCameraGlobalZ()` to change the camera's rotation about a specific inertial axis by a specified amount (given in degrees).
- Use `MoveCameraLocalX()`, `MoveCameraLocalY()`, and `MoveCameraLocalZ()` to move the camera a specified number of units along a given local axis.
- Use `GetCameraX()`, `GetCameraY()`, and `GetCameraZ()` to obtain the position of the camera.
- Use `GetCameraAngleX()`, `GetCameraAngleY()`, and `GetCameraAngleZ()` to discover the angle by which the camera is currently rotated about a given local axis.
- Use `GetObjectInScreen()` to discover if a specified object is currently visible on the screen.
- Using camera rotation and positioning, a first-person perspective can be achieved.
- 2D objects can appear to be 3D using billboard where a plane is kept at right angles to the camera.

Lights

Introduction

So far we have only looked at the AGK statement for creating a directional light. However, other commands exist to create point lights and manipulate the characteristics of both directional and point lights.

Directional Lights

SetLightDirectionalDirection()

To change the angle of the light produced by a directional light, use the `SetLightDirectionalDirection()` statement (see FIG-23.92).

FIG-23.92

SetLightDirectional
↪ Direction()

`SetLightDirectionalDirection` ((`id` , `x` , `y` , `z`))

where

id is an integer value giving the ID of an existing directional light.

x, y, z are real values defining the vector of the light rays.

SetLightDirectionalColor()

To change the colour of the light produced by a directional light, use the `SetLightDirectionalColor()` statement (see FIG-23.93).

FIG-23.93

SetLightDirectionalColor()

`SetLightDirectionalColor` ((`id` , `ir` , `ig` , `ib`))

where

id is an integer value giving the ID of the directional light.

ir, ig, ib are integer values (0 to 255) giving the intensity of the red, green and blue components of the light.

The program in FIG-23.94 demonstrates the use of both a change of direction and a change of colour as a directional light illuminates the surface of a sphere.

FIG-23.94

Using a Directional Light

```
rem *** Using Directional Light ***  
  
rem *** Create sphere ***  
CreateObjectSphere(1,5,20,20)  
  
rem *** Add directional light ***  
CreateLightDirectional(1,50,-15,10,150,150,150)  
  
rem *** Position camera ***  
SetCameraPosition(1,0,5,-20)  
SetCameraLookAt(1,0,0,0,0)
```



FIG-23.94

(continued)

Using a Directional Light

```

rem *** Light's y offset ***
yoff# = -15
rem *** Light's red component ***
red# = 150
do
    rem *** Change y offset ***
    yoff# = yoff#+0.1
    rem *** Change red intensity ***
    red# = red#+0.1
    rem *** Update direction and colour ***
    SetLightDirectionalDirection(1,50,yoff#,10)
    SetLightDirectionalColor(1,red#,150,150)
    Sync()
    rem *** Wait 10 msecs ***
    Sleep(10)
loop

```

Activity 23.38

Start a new project called *DLights3D* and implement the code given in FIG-23.94. Test and save your program.

GetLightDirectionalExists()

To check if a specified ID has been assigned to an existing directional light, use the `GetLightDirectionalExists()` statement (see FIG-23.95).

FIG-23.95`GetLightDirectionalExists()`

integer `GetLightDirectionalExists` ((id))

where

id is an integer value giving the ID to be checked.

If a directional light of that ID exists, the function returns 1, otherwise zero is returned.

DeleteLightDirectional()

To delete a specific directional light use `DeleteLightDirectional()` (see FIG-23.96).

FIG-23.96`DeleteLightDirectional()`

`DeleteLightDirectional` ((id))

where

id is an integer value giving the ID of the light to be deleted.

ClearLightDirectionals()

If you have several directional lights, you can delete them all with a single statement - `ClearLightDirectionals()` (see FIG-23.97).

FIG-23.97`ClearLightDirectionals()`

`ClearLightDirectionals` ()

CreateLightPoint()

The second type of light available in AGK is the point light. When you create a light of this type, you need to specify the position from which the light originates, its radius, and its colour. The further an object is from the light source, the weaker the effect of the light; if the object is beyond the specified radius, then the light will have no effect on it.

FIG-23.98

CreateLightPoint()

To create a point light, use `CreateLightPoint()` (see FIG-23.98).

`CreateLightPoint` ((`id` , `x` , `y` , `z` , `rad` , `ir` , `ig` , `ib`)

where

id	is an integer value giving the ID to be assigned to the point light.
x, y, z	are real values giving the coordinates at which the light is to be positioned.
rad	is a real value giving the square of the distance over which the light is at its full intensity. Hence, if you require the light to still be at full intensity 10 units from its source, <i>rad</i> would have a value of 100 (10^2). Outside this range, the light intensity drops as the distance increases.
ir, ig, ib	are integer values (0 to 255) giving the intensities of the red, green and blue components of the colour of the light emitted. Set all three to 255 for white light.

Activity 23.39

Reload *FP3D*. Change the brightness of the directional light to 10,10,10.

Add a point light at position (35,10,30) with a radius of 10 and a colour of 250, 250, 0. Run the program and observe the effect of the new light.

Save your program.

SetLightPointPosition()

To reposition an existing point light, use `SetLightPointPosition()` (see FIG-23.99).

FIG-23.99

SetLightPointPosition()

`SetLightPointPosition` ((`id` , `x` , `y` , `z`)

where

id	is an integer value giving the ID of the point light.
x, y, z	are real values giving the coordinates at which the light is to be repositioned.

Activity 23.40

Modify *FP3D*'s *HandleCamera()* function so that the point light is positioned on the camera at all times.

Run the program and observe the effect of the change. Save your program.

SetLightPointColor()

To change the colour of the light emitted by a point light, use `SetLightPointColor()` (see FIG-23.100).

FIG-23.100

SetLightPointColor()

SetLightPointColor ((id , ir , ig , ib))

where

id is an integer value giving the ID of the point light.

ir, ig, ib are integer values (0 to 255) giving the intensities of the red, green and blue components of the colour of the light emitted. Set all three to 255 for white light.

Activity 23.41

Load *FP3D*. Create a record structure giving details of the point light and a corresponding global variable using the following code:

```
rem *** Structure for point light data ***
type PointLightType
    colour                //250: yellow; 0: red
endtype
rem *** Global variables ***
rem *** Point light info ***
global light as PointLightType
light.colour = 1
```

Add a new function called *HandleLight()* which switches the value in *light.colour* between 250 and 0 and the colour of the point light between yellow (250,250,0) and red (250,0,0) when the *Enter* key is pressed.

The new function should be called from within the `do..loop` structure in the main section of the program.

Test and save your program.

SetLightPointRadius()

To change the sphere of the light's influence, use `SetLightPointRadius()` (see FIG-23.101).

FIG-23.101

SetLightPointRadius()

SetLightPointRadius ((id , rad))

where

id is an integer value giving the ID of the point light.

rad is a real value giving the new radius of the light.

Activity 23.42

In *FP3D*, modify the definition of *PointLightType* and *HandleLight()* so that the radius of the point light increases by 1 unit during each frame the + key is held down. Similarly, decrease the radius by 1 unit while the - key is held down. The radius should be restrained to lie within the limits 5 to 50.

Test and save your program.

GetLightPointExists()

To check if a specified ID has been assigned to an existing point light, use the `GetLightPointExists()` statement (see FIG-23.102).

FIG-23.102

GetLightPointExists() integer `GetLightPointExists` ((id))

where

id is an integer value giving the ID to be checked.

If a directional light of that ID exists, the function returns 1, otherwise zero is returned.

DeleteLightPoint()

To delete a specific point light use `DeleteLightPoint()` (see FIG-23.103).

FIG-23.103

DeleteLightPoint() `DeleteLightPoint` ((id))

where

id is an integer value giving the ID of the light to be deleted.

ClearLightPoints()

If you have several point lights, you can delete them all with a single statement - `ClearLightPoints()` (see FIG-23.104).

FIG-23.104

ClearLightPoints() `ClearLightPoints` (())

Object Reflectivity

SetObjectLightMode()

You have the option to make a specific object unreflective to any new lights you have created by using the `SetObjectLightMode()` statement (see FIG-23.105).

FIG-23.105

SetObjectLightMode() `SetObjectLightMode` ((id , imode))

where

id	is an integer value giving the ID of the object whose light reflecting properties are to be changed.
imode	is an integer value (0 or 1) giving the reflection mode (0: no reflection other than the default ambient light, 1: normal reflection of all lights).

Activity 23.43

In *FP3D* modify the floor (object 2) so that it is lit only by the default ambient light. Test and save your program.

When you set an object's light mode to zero, it will still reflect the original ambient light

Summary

- AGK 3D scenes default to using ambient light which creates light coming equally from all directions.
- Ambient lit objects show no form of shading.
- A directional light creates a light in which all rays are parallel and whose brightness does not diminish over distance.
- Use `SetLightDirectionalDirection()` to change the direction of an existing directional light.
- Use `SetLightDirectionalColor()` to change the colour of an existing directional light.
- Use `GetDirectionalLightExists()` to check if a directional light of a specified ID currently exists.
- Use `DeleteLightDirectional()` to delete a specific directional light.
- Use `ClearLightDirectionals()` to delete all existing directional lights.
- A point light creates light which originates from a specific point in space.
- The rays of a point light radiate equally in all directions from the source point.
- The intensity of the light from a point light decreases over distance.
- Use `CreateLightPoint()` to create a new point light.
- Use `SetPointLightPosition()` to reposition an existing point light.
- Use `SetLightPointColor()` to modify the colour of the light produced by an existing point light.
- Use `SetLightPointRadius()` to change the radius of the sphere within which the point light's rays can be detected.
- Use `GetPointLightExists()` to check if a point light of a specified ID currently exists.
- Use `DeleteLightPoint()` to delete a specific point light.
- Use `ClearLightPoints()` to delete all existing point lights.

Collisions

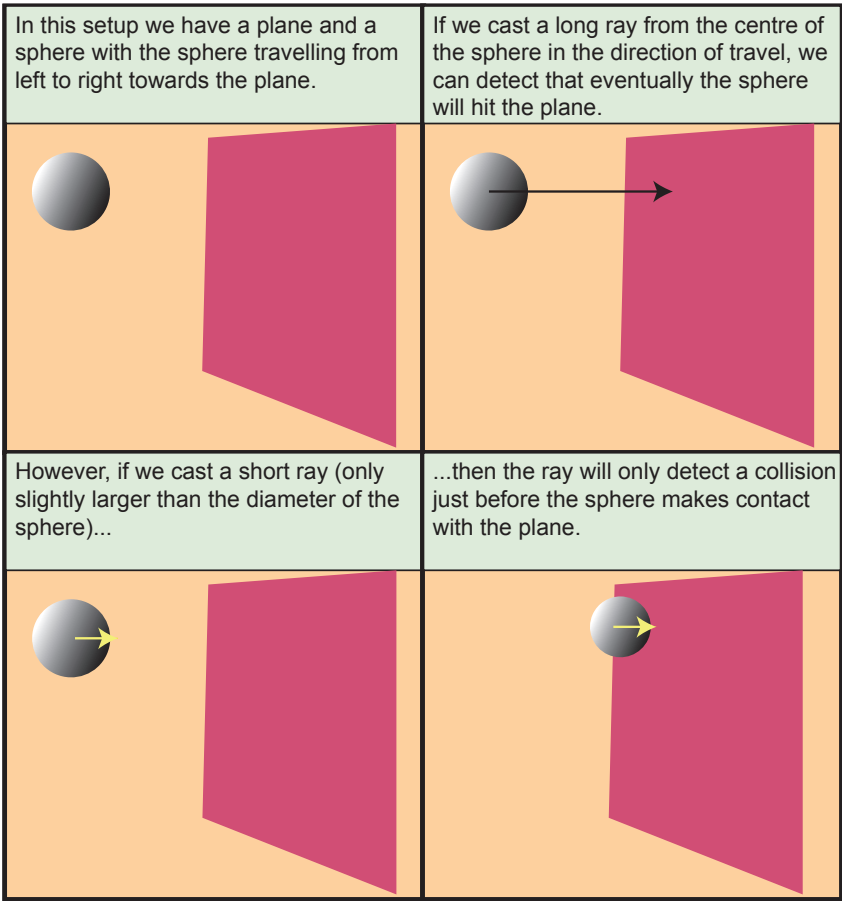
Introduction

Just as we needed to detect collisions between sprites, so we also need to detect collisions between 3D objects.

3D collision detection does not make use of bounding shapes as employed in 2D collision detection. To detect a collision between two 3D objects you must start by producing a ray cast from one object towards another. The ray cast can detect if another 3D object lies along its path. By making the length of the ray cast match the size and direction of the moving object, then it is possible to detect a collision between objects. FIG-23.106 shows the concepts involved.

FIG-23.106

Using a Ray Cast to
Detect a Collision



Ray Cast Statements

ObjectRayCast()

To cast a ray in 3D space, use the `ObjectRayCast()` statement. This will cast a ray between two points. You can either check for a specific object being hit or just check for any object being hit. You must also specify the start and end points of the ray.

The `ObjectRayCast()` statement's format is shown in FIG-23.107).

FIG-23.107 ObjectRayCast()

`ObjectRayCast ((id , x1 , y1 , z1 , x2 , y2 , z2)`

where

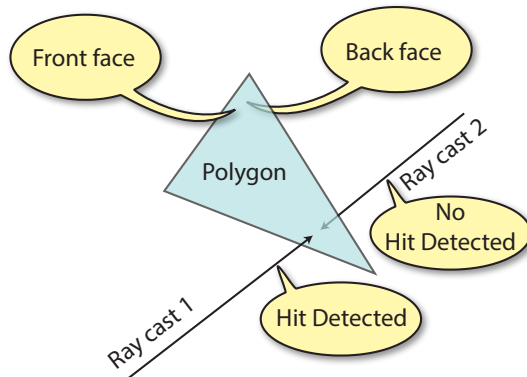
- | | |
|-------------------|---|
| id | is an integer value giving the ID of the object to be checked or zero if all objects are to be checked. |
| x1, y1, z1 | are real numbers giving the coordinates of the starting point of the ray. |
| x2, y2, z2 | are real numbers giving the coordinates of the end point of the ray. |

If you have specified the ID to be checked, the function will return 1 for a hit and zero for a miss. If you have not specified a specific object to be checked (by setting *id* to zero), the function will return the ID of the first object encountered by the ray or zero if none are encountered.

The function only detects collisions with the front face of a polygon; back face collisions are not detected (see FIG-23.108).

FIG-23.108

Ray Casts Detect only
Front Face Collisions



The program in FIG-23.109 creates two objects: a plane and a sphere. The sphere is moving on a trajectory which will intersect the plane. A ray cast is performed starting at the centre of the sphere and ending 15 units further along its trajectory. The result of that cast is then displayed on the screen.

FIG-23.109

Using Ray Casting

```
rem *** 3D ray casting ***

rem *** Create plane ***
CreateObjectPlane(1,15,15)
rem *** Colour plane ***
SetObjectColor(1,200,100,100,255)
rem *** Move plane back ***
SetObjectPosition(1,0,0,15)

rem *** Create sphere ***
CreateObjectSphere(2,3,15,15)
rem *** Position sphere ***
SetObjectPosition(2,-2,5,-10)
```



FIG-23.109

(continued)

Using Ray Casting

```

rem *** Set up light ***
CreateLightDirectional(1,10,-10,20,200,200)

rem *** Position camera ***
SetCameraPosition(1,20,5,-10)
SetCameraLookAt(1,0,0,5,0)

rem *** Create text to display result ***
CreateText(1,"")

rem *** Sphere's initial position on the z-axis ***
z# = -10

do
    rem *** Ray from centre of sphere 15 ***
    rem *** units along the z-axis ***
    hit = ObjectRayCast(1,-2,5,z#,-2,5,z#+15)
    rem *** Display result ***
    SetTextString(1,Str(hit))
    rem *** Move sphere 0.1 units along z-axis ***
    z# = z# + 0.1
    SetObjectPosition(2,-2,5,z#)
    rem *** Update display ***
    Sync()
loop

```

Activity 23.44

Start a new project, *RayCast3D*, (size 900 x 900) and implement the code given in FIG-23.109.

Note how far the sphere is from the plane when a hit is detected. Modify the code so that the ray is only 1.5 units long (the radius of the sphere).

Test and save your program.

Because the sphere in the program above is moving parallel to the z-axis, it is quite simple to calculate the end point of the ray cast. However, when an object moves in a direction not parallel to an axis, then we need to do a bit more in the way of calculation to determine that end point.

If the trajectory of the sphere moves it 0.025 units in the x-direction, -0.05 units in the y direction and 0.2 units in the z direction on each frame, then we could move the sphere using the following code:

```

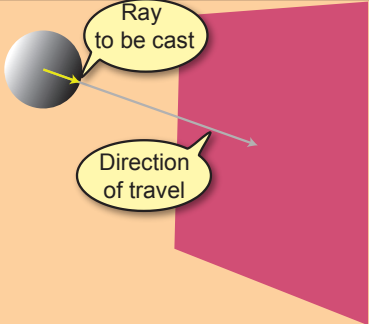
rem *** Sphere's initial position ***
x# = -2
y# = 5
z# = -10
do
    rem *** Move sphere along trajectory ***
    x# = x# + 0.025
    y# = y# - 0.05
    z# = z# + 0.2
    SetObjectPosition(2,x#,y#,z#)
    rem *** Update display ***
    Sync()
loop

```

We know the starting point of the ray cast is at the centre of the sphere ($x\#, y\#, z\#$) and that the end point should be a distance of 1.5 units from the start point. All we need now is to calculate the coordinates of that end point. The steps involved in arriving at the end coordinates are shown in FIG-23.110.

FIG-23.110

Calculating the Offsets
for a Ray Cast

<p>The ray being cast follows the same direction as the sphere.</p>	<p>In a single frame the sphere travels 0.025 in the x direction, -0.05 in the y direction and 0.2 in the z direction giving a distance of...</p>
	<p>distance travelled =</p> $\sqrt{0.025^2 + (-0.05)^2 + 0.2^2}$ $= \sqrt{0.000625 + 0.0025 + 0.04}$ $= \sqrt{0.043125}$ $= 0.207666$
<p>So the distance travelled in a single frame is 0.207666. If the distance travelled were 1.5 units then we would need to multiply the first figure by...</p>	<p>When the distance travelled in one frame is multiplied by 7.223137 to give a total distance travelled of 1.5, the distance travelled along each axis can be found by multiplying the original distances by the same figure.</p>
<p>multiplier = $1.5 / 0.207666$</p> <p>= 7.223137</p>	<p>distance along x = $0.025 * 7.223137$</p> <p>= 0.180578</p> <p>distance along y = $-0.05 * 7.223137$</p> <p>= -0.361157</p> <p>distance along z = $0.2 * 7.223137$</p> <p>= 1.444627</p>

With the distances travelled in each direction along a line which is 1.5 units in length, we can calculate the endpoint coordinates as:

```
endx# = x# + distx#
endy# = y# + disty#
endz# = z# + distz#
```

An updated version of the previous program is given in FIG-23.111. This calculates the end point of the ray using the direction of movement.

FIG-23.111

Using Ray Casting -
Update

```
rem *** 3D Ray casting ***

rem *** Create first plane ***
CreateObjectPlane(1,15,15)
rem *** Colour plane ***
SetObjectColor(1,200,100,100,255)

rem *** Move plane back ***
SetObjectPosition(1,0,0,15)
```



FIG-23.111

(continued)

Using Ray Casting -
Update

```

rem *** Create sphere ***
CreateObjectSphere(2,3,15,15)
rem *** Position sphere ***
SetObjectPosition(2,-2,5,-10)

rem *** Set up light ***
CreateLightDirectional(1,10,-10,20,200,200,200)
rem *** Position camera ***
SetCameraPosition(1,20,5,-10)
SetCameraLookAt(1,0,0,5,0)
rem *** Create text to display result ***
CreateText(1,"")

rem *** Sphere's initial position on all axes ***
x# = -2
y# = 5
z# = -10

rem *** Movement along each axis per frame ***
xstep# = 0.025
ystep# = -0.05
zstep# = 0.2

rem *** Calculate movement along each axis over a distance of 1.5
units ***
multiplier# = 1.5 / (Sqrt(xstep#^2 + ystep#^2 + zstep#^2))
xdist# = xstep# * multiplier#
ydist# = ystep# * multiplier#
zdist# = zstep# * multiplier#

do
  rem *** Ray from centre of sphere 1.5 units ***
  hit = ObjectRayCast(1,x#,y#,z#, x#+xdist#, y#+ydist#,
    ↵ z#+zdist#)
  rem *** Display result ***
  SetTextString(1,Str(hit))
  rem *** Move sphere along trajectory ***
  x# = x# + xstep#
  y# = y# + ystep#
  z# = z# + zstep#
  SetObjectPosition(2,x#,y#,z#)
  rem *** Update display ***
  Sync()
loop

```

Activity 23.45

Modify *RayCast3D* to match the code given in FIG-23.111. Test and save your program.

As a general rule, we want objects to appear solid and not to pass through each other. One way to do this is to allow movement only when the ray cast from the object does not detect another object in its path. For example, we could move our sphere conditionally with the following code:

```

rem *** If no collision, move sphere ***
if hit = 0

```

```

rem *** Move sphere along trajectory ***
x# = x# + xstep#
y# = y# + ystep#
z# = z# + zstep#
SetObjectPosition(2,x#,y#,z#)
endif

```

Activity 23.46

Modify *RayCast3D* so that the sphere stops moving when it reaches the plane.

Test and save your program.

SetObjectCollisionMode()

If you don't want a specific 3D object to be detectable during a ray cast operation, use `SetObjectCollisionMode()` (see FIG-23.112).

FIG-23.112

SetObjectCollisionMode()

`SetObjectCollisionMode (id , imode)`

where

id is an integer value giving the ID of the object whose collision mode is to be set.

imode is an integer value (0 or 1) which is used to set the object's collision mode (0: object not detectable by a cast; 1: object detectable).

Activity 23.47

Modify *RayCast3D* so that the plane is not detectable.

Run the program and observe the result of this change.

Do NOT save this version of the program.

GetObjectRayCastX(), GetObjectRayCastY() and GetObjectRayCastZ()

To discover the exact point at which a cast collides with an object, use `GetObjectRayCastX()`, `GetObjectRayCastY()`, and `GetObjectRayCastZ()` (see FIG-23.113).

FIG-23.113

GetObjectRayCastX()
GetObjectRayCastY()
GetObjectRayCastZ()

float `GetObjectRayCastX (idx)`

float `GetObjectRayCastY (idx)`

float `GetObjectRayCastZ (idx)`

where

idx

is an integer value (0 to 3) giving the index of the hit being interrogated.

For the moment we will use an *idx* value of zero. The purpose of the other options for this parameter will be explained shortly.

Activity 23.48

Modify *RayCast3D* so that the text displays the coordinates of the collision when the ray cast detects a hit.

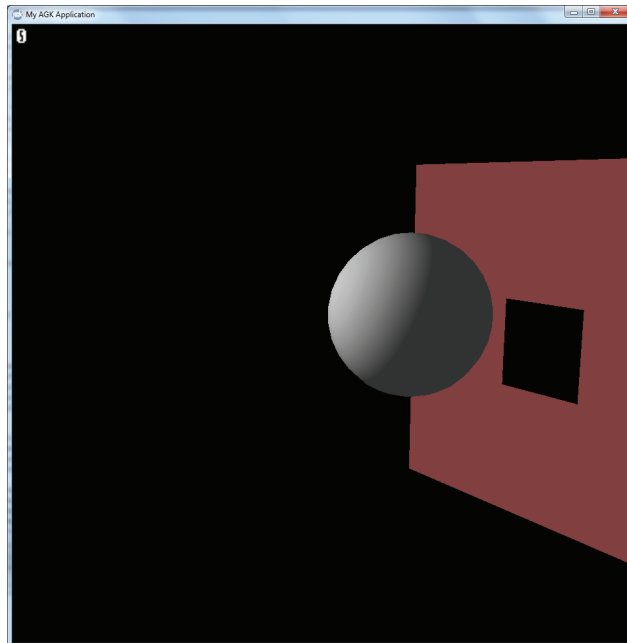
Test and save your program.

ObjectSphereCast()

Casting a ray to detect a collision will not always give an accurate result. For example, looking at the setup shown in FIG-23.114 we can see that the sphere is too large to fit through the hole in the plane.

FIG-23.114

Ray Casting Problems



However, since the trajectory of the sphere passes through the middle of the hole, the ray cast will not detect the plane and allow the sphere to continue on its way.

Activity 23.49

Modify *RayCast3D* so that the model *HoledPlane.obj* is used in place of the program-created plane. Remember to copy the file into the project's *media* folder.

Change the diameter of the sphere from 3 to 6 in the `CreateObjectSphere()` statement. This will require a change to the length of the ray cast from 1.5 to 3.0 units. Change the appropriate line within the code.

Test and save your program. Does the sphere pass through the plane?

Because the ray cast has no width, it does not detect the fact that the sphere is too wide to go through the hole. To overcome this problem, we can replace the ray cast with a sphere cast. A sphere cast gives a volume to a cast, creating a cylinder-shaped cast (see FIG-23.115).

FIG-23.115

The Advantage of
Sphere Casting

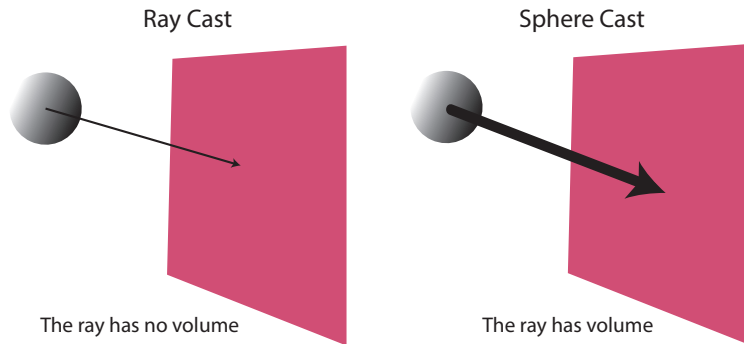


FIG-23.116

ObjectSphereCast()

To create a sphere cast, use the `ObjectSphereCast()` statement (see FIG-23.116).

```
integer ObjectSphereCast ( ( id , x1 , y1 , z1 , x2 , y2 , z2 , rad ) )
```

where

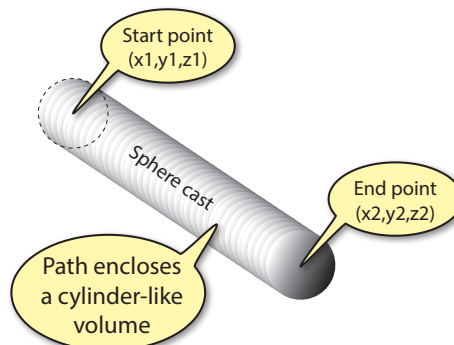
- | | |
|-------------------|---|
| id | is an integer value giving the ID of the object to be checked or zero if all objects are to be checked. |
| x1, y1, z1 | are real numbers giving the coordinates of the starting point of the sphere cast. |
| x2, y2, z2 | are real numbers giving the coordinates of the end point of the sphere cast. |
| rad | is a real number giving the radius of the sphere cast. |

If you have specified the ID to be checked, the function will return 1 for a hit and zero for a miss. If you have not specified a specific object to be checked (by setting *id* to zero), the function will return the ID of the first object encountered by the ray or zero if no object is encountered.

The path being produced by this command can be thought of as enclosing a cylinder-like shape (but with curved top and bottom ends). This shape is created by the movement of a sphere of a given radius from (*x1*, *y1*, *z1*) to (*x2*, *y2*, *z2*) (see FIG-23.117).

FIG-23.117

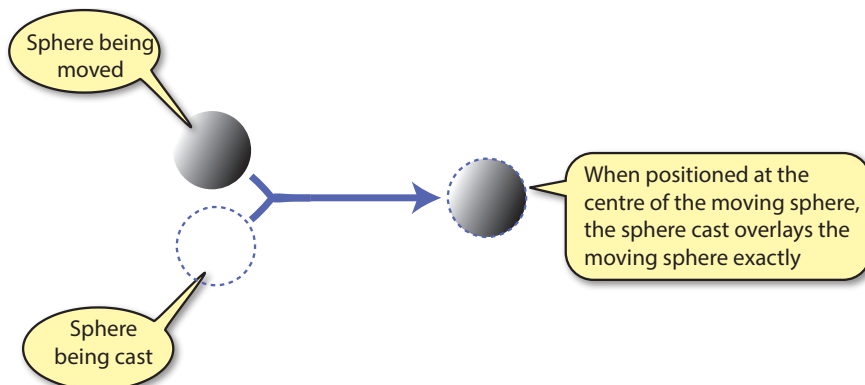
Visualisation of a Sphere
Cast



To check if the sphere can pass through the hole in the plane shown in FIG-23.114, we would make the radius of the sphere cast match that of the moving sphere object. However, because the sphere used in the cast has volume, we must think carefully about the end point of the sphere cast. When we created a ray cast, we wanted the ray to reach from the centre of the moving sphere to its surface. But when using a sphere cast with a radius matching that of the moving sphere, it might seem that the start and end points of the cast should be equal (see FIG-23.118).

FIG-23.118

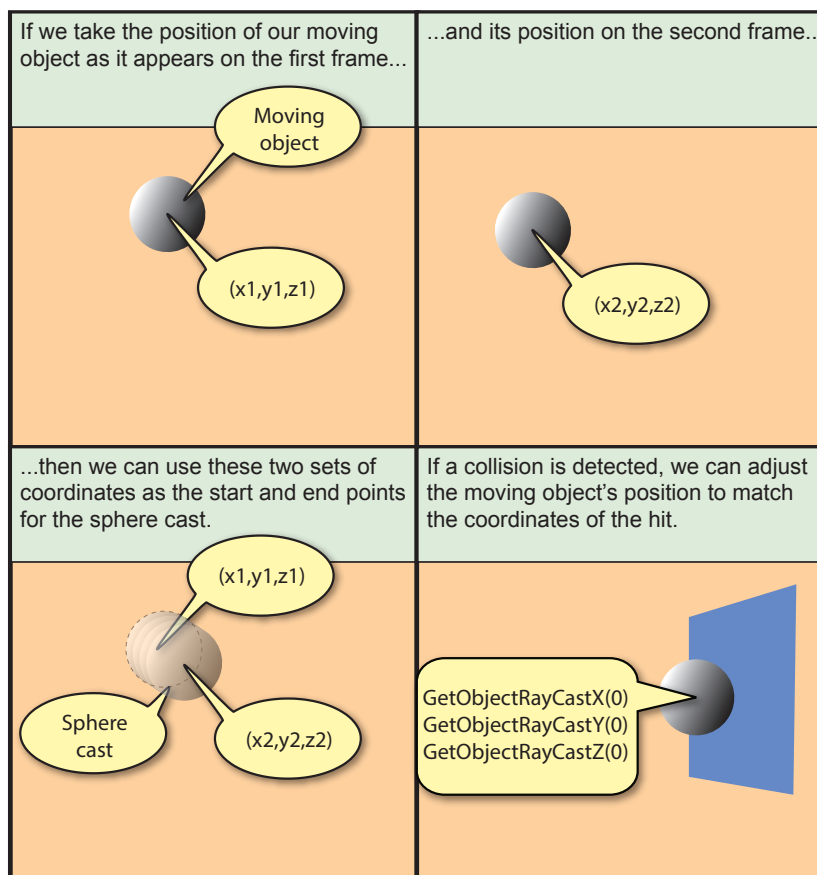
Overlapping An Object and a Sphere Cast



However, no cast can have its start and end points equal to each other. If we were to do this, then no information is given about the direction in which the cast is travelling. This direction information is required by AGK in order to handle certain collision situations. To overcome this, we need to use the previous and current positions of the moving object as the start and end points for the sphere cast (see FIG-23.119).

FIG-23.119

Calculating a Sphere Cast's Start and End Points



Another problem to watch out for is that a sphere cast can, on occasion, return a hit on the object that is being moved and for this reason, it is best to turn off collision detection for that object.

The program in FIG-23.120 is a variation on the earlier ray cast program but makes use of a sphere cast to stop a moving sphere object passing through a holed plane.

FIG-23.120

Using a Sphere Cast

```
rem *** 3D Sphere Casting ***

rem *** Load holed plane ***
LoadObject(1,"HoledPlane.obj",15)
rem *** Colour plane ***
SetObjectColor(1,200,100,100,255)
rem *** Move plane back ***
SetObjectPosition(1,0,0,15)

rem *** Create sphere ***
CreateObjectSphere(2,6,15,15)
rem *** Position sphere ***
x# = -2
y# = 5
z# = -10
SetObjectPosition(2,x#,y#,z#)

rem *** Set up light ***
CreateLightDirectional(1,10,-10,20,200,200)

rem *** Position camera ***
SetCameraPosition(1,20,5,-10)
SetCameraLookAt(1,0,0,5,0)

rem *** Create text to display result ***
CreateText(1,"")
SetTextSize(1,2.5)

rem *** Set sphere's movement along each axis per frame ***
xstep# = 0.025
ystep# = -0.05
zstep# = 0.2

rem *** The cast cannot hit the moving sphere ***
SetObjectCollisionMode(2,0)

rem *** No collision yet ***
hit = 0

do
  rem *** if no collision yet ***
  if hit = 0
    rem *** Get current position of sphere object ***
    oldx# = GetObjectX(2)
    oldy# = GetObjectY(2)
    oldz# = GetObjectZ(2)

    rem *** Move the sphere object ***
    MoveObjectLocalX(2,xstep#)
    MoveObjectLocalY(2,ystep#)
    MoveObjectLocalZ(2,zstep# )

    rem *** Get sphere's new position ***
    x# = GetObjectX(2)
```



FIG-23.120

(continued)

Using a Sphere Cast

```

y# = GetObjectY(2)
z# = GetObjectZ(2)
rem *** Perform sphere cast between old and new positions
↳***
hit = ObjectSphereCast(0,oldx#,oldy#,oldz#,x#,y#,z#,1.5)
rem *** If hit...
if hit > 0
    rem *** Reposition the sphere at the point of collision
    ↳***
    SetObjectPosition(2, GetObjectRayCastX(0),
    ↳GetObjectRayCastY(0),GetObjectRayCastZ(0) )
    rem *** Give details of object hit ***
    SetTextString(1,"Hit object " +
    ↳Str(GetObjectRayCastHitID(0))
    ↳+" at ("(Str(GetObjectRayCastx(0)," " +
    ↳Str(GetObjectRayCastY(0)) + "," +
    ↳Str(GetObjectRayCastZ(0))+ ")")
endif
endif
Sync()
loop

```

Notice that the `GetObjectRayCast()` functions also operate correctly when retrieving the collision point of a sphere cast.

Activity 23.50

Start a new project called *SphereCast3D* and implement the code given in FIG-23.120. Copy the file *HoledPlane.obj* to the project's *media* folder.

Test and save your program.

Activity 23.51

Reload *FP3D* and position a sphere cast around the camera. Make the diameter of the sphere being cast 1.5 units.

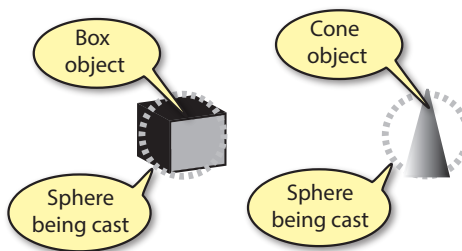
Modify the camera's movement so it cannot pass through the walls of the maze.

Test and save your program.

Of course, a sphere cast works best when the object it is linked to is itself a sphere, since the cast and object can occupy exactly the same volume. However, you can still use a sphere cast to detect when moving objects of other shapes collide; you just have to accept a compromise in the accuracy of the collision detection (see FIG-23.121).

FIG-23.121

Sphere Casts for Other Primitives



ObjectSphereSlide()

In the real world, when a moving object strikes some other immovable object it will be deflected from its course and may strike yet more objects. If the situation you are trying to achieve in your program is of this nature, then you can make use of the `ObjectSphereSlide()` statement which will generate details of up to three collisions as the moving object deflects off other elements in the scene. The statement has the same format as the earlier `ObjectSphereCast()`, but generates more information which can then be accessed by other collision-related statements.

FIG-23.122

`ObjectSphereSlide()`

The format for `ObjectSphereSlide()` is given in FIG-23.122.

integer `ObjectSphereSlide` ((`id` , `x1` , `y1` , `z1` , `x2` , `y2` , `z2` , `rad`))

where

id	is an integer value giving the ID of the object to be checked or zero if all objects are to be checked.
x1, y1, z1	are real numbers giving the coordinates of the starting point of the sphere slide.
x2, y2, z2	are real numbers giving the coordinates of the end point of the sphere slide.
rad	is a real number giving the radius of the sphere used.

GetObjectRayCastNumHits()

To discover the number of objects hit by a cast or slide, use the statement `GetObjectRayCastNumHits()` (see FIG-23.123).

FIG-23.123

`GetObjectRayCastNumHits()`

integer `GetObjectRayCastNumHits` (())

If you have used a ray or sphere cast, then the value returned will be either 0 (no hit) or 1 (hit), but when a sphere slide has been used, the value returned can lie between 0 and 3.

GetObjectRayCastHitID()

If you need to discover the ID of the 3D object hit by a cast or slide, use `GetObjectRayCastHitID()` (see FIG-23.124).

FIG-23.124

`GetObjectRayCastHitID()`

integer `GetObjectRayCastHitID` ((`idx`))

where

idx	is an integer value (0 to 2) giving the index of the hit item. This index derives from the number of hits reported. In the case of a cast, a maximum of one object can be hit, so the <i>idx</i> value should be zero; when a slide has been used, the maximum of three hits means <i>idx</i> can be as high as 2 (assuming three items have been hit).
------------	---

GetObjectRayCastSlideX(), GetObjectRayCastSlideY() and GetObjectRayCastSlideZ()

When a collision is detected using a sliding sphere cast, you should reposition the moving object using the statements `GetObjectRayCastSlideX()`, `GetObjectRayCastSlideY()`, and `GetObjectRayCastSlideZ()` (see FIG-23.125).

FIG-23.125

`GetObjectRayCastSlideX()`
`GetObjectRayCastSlideY()`
`GetObjectRayCastSlideZ()`

```
float GetObjectRayCastSlideX ( ( idx ) )  
float GetObjectRayCastSlideY ( ( idx ) )  
float GetObjectRayCastSlideZ ( ( idx ) )
```

where

idx is an integer value (0 only).

By placing the `ObjectSphereSlide()` and `GetObjectRayCastSlide()` statements within a loop, you can create a realistic movement of a moving object as it is deflected off one object to another. The effect will halt if three objects are hit.

The program in FIG-23.126 demonstrates the sliding effect with a moving sphere being deflected off three planes.

FIG-23.126

Using Sphere Slide

```
rem *** Demonstrating a Sphere Slide ***  
  
rem *** Create first wall ***  
CreateObjectPlane(1,20,10)  
rem *** Create second wall at right angles to first ***  
CreateObjectPlane(2,20,10)  
RotateObjectLocalY(2,90)  
SetObjectPosition(2,10,0,-10)  
rem *** Create floor ***  
CreateObjectPlane(3,20,20)  
RotateObjectLocalX(3,90)  
SetObjectPosition(3,0,-5,-10)  
  
rem *** Create directional light ***  
CreateLightDirectional(1,5,-10,0,255,255,255)  
  
rem *** Position Camera ***  
SetCameraPosition(1,-10,5,-50)  
SetCameraLookAt(1,10,0,0,0)  
  
rem *** Create sphere ***  
CreateObjectSphere(4,3,15,15)  
  
rem *** Sphere's position ***  
x# = 0  
y# = 4  
z# = -10  
SetObjectPosition(4,x#,y#,z#)  
  
rem *** Sphere's movement vector offsets ***  
xoff# = 0.15  
yoff# = -0.1  
zoff# = 0.025
```



FIG-23.126

(continued)

Using Sphere Slide

```

rem *** Switch off the sphere's collision detection ***
SetObjectCollisionMode(4,0)

rem *** Create text to display details of collision ***
CreateText(1,"")
SetTextSize(1,3)

do
    rem *** Get current position of sphere object ***
    oldx# = GetObjectX(4)
    oldy# = GetObjectY(4)
    oldz# = GetObjectZ(4)

    rem *** Move the sphere object ***
    MoveObjectLocalX(4,xoff#)
    MoveObjectLocalY(4,yoff#)
    MoveObjectLocalZ(4,zoff# )

    rem *** Get sphere's new position ***
    x# = GetObjectX(4)
    y# = GetObjectY(4)
    z# = GetObjectZ(4)

    rem *** Perform sphere slide between old and new position ***
    hit = ObjectSphereSlide(0,oldx#,oldy#,oldz#,x#,y#,z#,1.5)
    rem *** If hit...
    if hit > 0
        rem *** Reposition the sphere at the point of collision ***
        SetObjectPosition(4, GetObjectRayCastSlideX(0),
        ↵GetObjectRayCastSlideY(0), GetObjectRayCastSlideZ(0) )
        rem *** Display collision details ***
        SetTextString(1,"Hits: "+Str(GetObjectRayCastNumHits())+
        ↵" Object: "+Str(GetObjectRayCastHitID(0))+
        ↵" (" +Str(GetObjectRayCastSlideX(0))+" "+
        ↵Str(GetObjectRayCastSlideY(0))+" "+
        ↵Str(GetObjectRayCastSlideZ(0))+"")
        endif
        Sync()
    loop

```

Activity 23.52

Start a new project called *SphereSlide3D* and implement the code given in FIG-23.126.

Run the program. How many hits occur before the sphere stops moving?

Save your program.

GetObjectRayCastBounceX(), GetObjectRayCastBounceY() and GetObjectRayCastBounceZ()

To find the new trajectory of a moving object when it collides with another object, use the statements `GetObjectRayCastBounceX()`, `GetObjectRayCastBounceY()` and `GetObjectRayCastBounceZ()` (see FIG-23.127).

FIG-23.127

GetObjectRayCastBounceX()
 GetObjectRayCastBounceY()
 GetObjectRayCastBounceZ()

float **GetObjectRayCastBounceX** ((idx))
 float **GetObjectRayCastBounceY** ((idx))
 float **GetObjectRayCastBounceZ** ((idx))

where

idx is an integer value (0 to 2) giving the index of the hit being interrogated (0 only for ray and sphere casts; 0 to 2 for sliding sphere casts).

These functions return the offsets of the moving object's new trajectory.

The program in FIG-23.128 is a modification of the previous program (*SphereSlide3D*) which makes use of the bounce data to adjust the trajectory of the moving sphere as it hits each of the three walls.

FIG-23.128

Using Ray Cast Bounce

```
rem *** Demonstrating Bounce ***

rem *** Create first wall ***
CreateObjectPlane(1,20,10)
rem *** Create second wall at right angles to first ***
CreateObjectPlane(2,20,10)
RotateObjectLocalY(2,90)
SetObjectPosition(2,10,0,-10)
rem *** Create floor ***
CreateObjectPlane(3,20,20)
RotateObjectLocalX(3,90)
SetObjectPosition(3,0,-5,-10)

rem *** Create directional light ***
CreateLightDirectional(1,5,-10,0,255,255,255)

rem *** Position Camera ***
SetCameraPosition(1,-10,5,-50)
SetCameraLookAt(1,10,0,0,0)

rem *** Create sphere ***
CreateObjectSphere(4,3,15,15)

rem *** sphere's position ***
x# = -5
y# = 4
z# = -15
SetObjectPosition(4,x#,y#,z#)

rem *** Sphere's movement vector offsets ***
xoff# = 0.075
yoff# = -0.05
zoff# = 0.15

rem *** Switch off the sphere's collision detection ***
SetObjectCollisionMode(4,0)

do
  rem *** Get current position of sphere object ***
```



FIG-23.128

(continued)

Using Ray Cast Bounce

```

    oldx# = GetObjectX(4)
    oldy# = GetObjectY(4)
    oldz# = GetObjectZ(4)

    rem *** Move the sphere object ***
    MoveObjectLocalX(4,xoff#)
    MoveObjectLocalY(4,yoff#)
    MoveObjectLocalZ(4,zoff# )

    rem *** Get sphere's new position ***
    x# = GetObjectX(4)
    y# = GetObjectY(4)
    z# = GetObjectZ(4)

    rem *** Perform sphere cast between old and new position ***
    hit = ObjectSphereCast(0,oldx#,oldy#,oldz#,x#,y#,z#,1.5)
    rem *** If hit...
    if hit > 0
        rem *** Reposition the sphere at the point of collision ***
        SetObjectPosition(4, GetObjectRayCastX(0),
        ↳GetObjectRayCastY(0), GetObjectRayCastZ(0))
        rem *** Change sphere's trajectory ***
        xoff# = GetObjectRayCastBounceX(0)
        yoff# = GetObjectRayCastBounceY(0)
        zoff# = GetObjectRayCastBounceZ(0)
    endif
    Sync()
loop

```

Activity 23.53

Modify *SphereSlide3D* to match the code given in FIG-23.128.

Run the program and observe the change in the sphere's trajectory as it hits each of the three surfaces. Save your program.

GetObjectRayCastNormalX(), GetObjectRayCastNormalY() and GetObjectRayCastNormalZ()

A collision normal is a vector which is at right-angles to the surface being hit by a cast or slide operation. (see FIG-23.129).

FIG-23.129

A Collision Normal

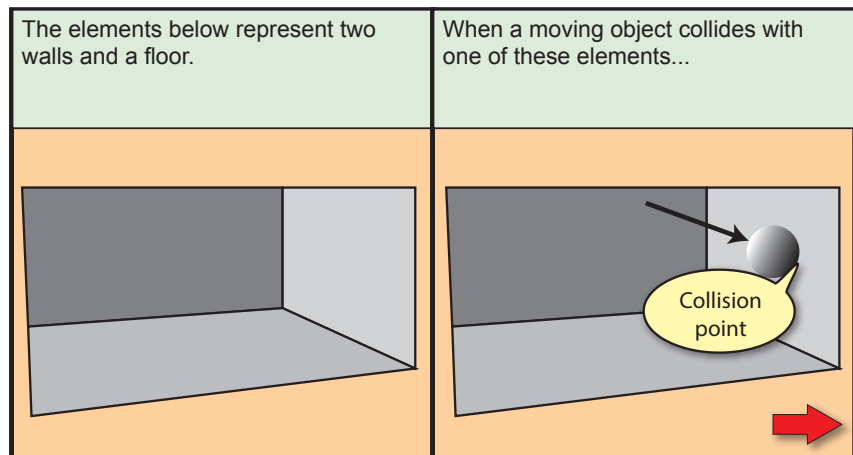
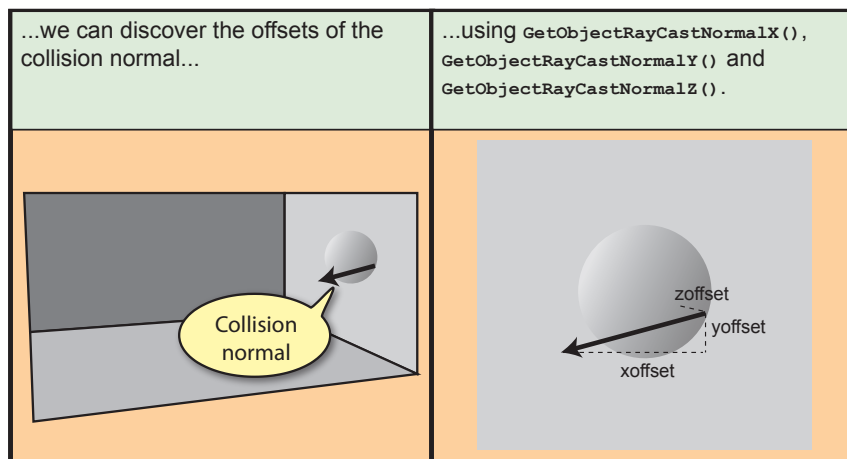


FIG-23.129

(continued)

A Collision Normal



The format for the statements `GetObjectRayCastNormalX()`, `GetObjectRayCastNormalY()` and `GetObjectRayCastNormalZ()` are shown in FIG-23.130).

FIG-23.130

`GetObjectRayCastNormalX()`
`GetObjectRayCastNormalY()`
`GetObjectRayCastNormalZ()`

```
float GetObjectRayCastNormalX ( ( idx ) )
float GetObjectRayCastNormalY ( ( idx ) )
float GetObjectRayCastNormalZ ( ( idx ) )
```

where

idx

is an integer value (0 to 2) giving the index of the hit being interrogated (0 only for ray and sphere casts; 0 to 2 for sliding sphere casts).

The collision normal information becomes useful when we want to place another object at the point of collision. For example, if we wanted the sphere in the last program to leave a mark at the point on each surface where a hit occurs, we would start by placing the required image on a plane. This plane could then be positioned at the point of collision. But the most important part is to orientate that plane so that it lies flat to the surface being hit. This can be done with the help of the normal data (see FIG-23.131).

FIG-23.131

Using a Collision Normal

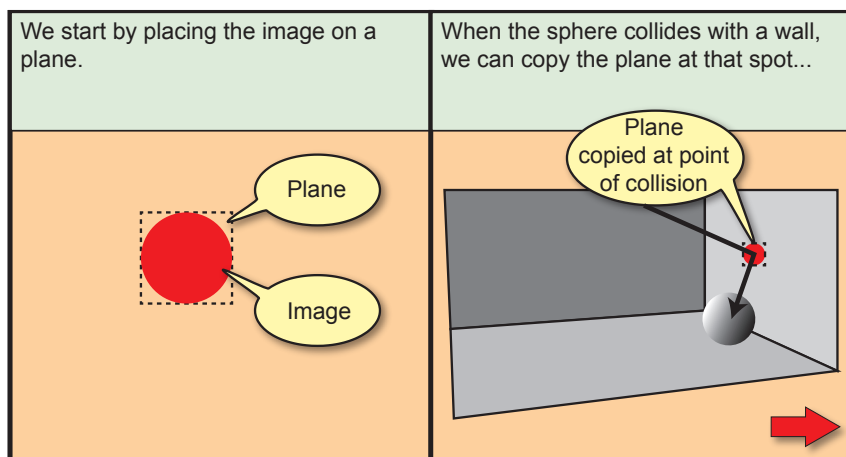
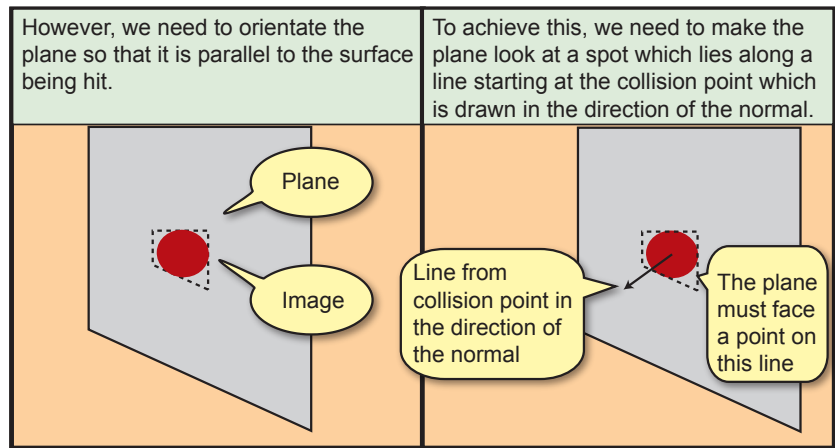


FIG-23.131

(continued)

Using a Collision Normal



In order to modify *SphereSlide3D* to leave a “mark” at each collision point, we need the following elements of code.

First we need to create a plane textured with the required image:

```
rem *** Create Bounce Mark plane ***
LoadImage(1,"RedCircle.png")
CreateObjectPlane(5,1.5,1.5)
SetObjectImage(5,1,0)
SetObjectTransparency(5,1)
SetObjectCollisionMode(5,0)
SetObjectPosition(5,0,1200,0)
```

Notice that the plane’s collision detection has been switched off and that it has been positioned far outside the scene captured by the camera. We will clone this plane at the appropriate position each time the sphere collides with a wall or the floor.

We need a variable to record the object ID being assigned to the cloned object:

```
rem *** Object ID ***
objno = 5
```

The variable starts at 5 - the ID of the new plane - and will be incremented when a new copy is created.

When a hit is detected we need to position and orientate a clone of the plane containing the bounce mark. To do this, the code is rewritten as:

```
if hit > 0
    rem *** Create a bounce mark plane ***
    inc objno
    InstanceObject(objno,5)
    rem *** Get the collision point ***
    hitx# = GetObjectRayCastX(0)
    hity# = GetObjectRayCastY(0)
    hitz# = GetObjectRayCastZ(0)
    rem *** Position bounce mark at collision point ***
    SetObjectPosition(objno,hitx#,hity#,hitz#)
    rem *** Orientate bounce mark along the normal ***
    ox# = hitx# + GetObjectRayCastNormalX(0)
    oy# = hity# + GetObjectRayCastNormalY(0)
    oz# = hitz# + GetObjectRayCastNormalZ(0)
```

```

SetObjectLookat(objno,ox#,oy#,oz#,0)
rem *** Reposition sphere at the point of collision ***
SetObjectPosition(4,hitx#,hity#,hitz#)
rem *** Change sphere's trajectory ***
xoff# = GetObjectRayCastBounceX(0)
yoff# = GetObjectRayCastBounceY(0)
zoff# = GetObjectRayCastBounceZ(0)
endif

```

Activity 23.54

Using the code given above, modify *SphereSlide3D* so that a bounce mark is created at each position where a collision occurs. (Copy *RedCircle.png* to the *media* folder.)

Run the program and observe the results. Save your program.

GetObjectRayCastDistance()

To discover the distance between the start point of a cast and the point at which a collision is detected, use `GetObjectRayCastDistance()` (see FIG-23.132).

FIG-23.132

`GetObjectRayCastDistance()` float `GetObjectRayCastDistance` ((`idx`))

where

idx is an integer value (0 to 2) giving the index of the hit being interrogated (0 only for ray and sphere casts; 0 to 2 for sliding sphere casts).

Summary

- Ray casting involves creating an imaginary line between two points in 3D space.
- Use `ObjectRayCast()` to create a ray cast between two points. The function can be used to detect if a specific object has been encountered along the path of the ray cast or return the ID of the first object encountered.
- A ray cast will only detect collisions with the front faces of an object, not the back faces.
- Use `SetObjectCollisionMode()` to stop an object being detected by a ray cast.
- Use `GetObjectRayCastX()`, `GetObjectRayCastY()`, and `GetObjectRayCastZ()` to determine the point at which the ray cast hits the reported object.
- Use `ObjectSphereCast()` to cast a sphere along a specified line.
- The path of a sphere cast has a cylinder-shaped volume; a ray cast has no volume.
- Sphere casts can be used as a bounding volume for a 3D shape though for any shape other than a sphere, that bounding volume is only a rough approximation of the true shape.

- Use `ObjectSphereSlide()` to create a sliding effect, moving an object along the surface of up to three objects.
- Use `GetObjectRayCastNumHit()` to discover the number of objects hit by a cast. Ray and sphere casts will hit a maximum of 1 object, a sphere slide can hit up to 3 objects.
- Use `GetObjectRayCastHitID()` to discover the ID of the object hit by a ray or sphere cast. In the case of a sphere slide, up to three IDs can be accessed by setting the parameter to a value from 0 to 2.
- Use `GetObjectRayCastSlideX()`, `GetObjectRayCastSlideY()`, and `GetObjectRayCastSlideZ()` to determine the slide point to which a moving object should be placed after a hit occurs when using the `ObjectSphereSlide()` statement.
- Use `GetObjectRayCastBounceX()`, `GetObjectRayCastBounceY()`, and `GetObjectRayCastBounceZ()` to find the offsets required to create a bounce effect after using a ray or sphere cast.
- Use `GetObjectRayCastNormalX()`, `GetObjectRayCastNormalY()`, and `GetObjectRayCastBounceZ()` to find the offsets of the normal vector to the surface hit by a ray or sphere cast. For a sphere slide, set the parameter to a value between 0 and 2 for the required object involved.
- Use `GetObjectRayCastDistance()` to discover the distance along a ray or sphere cast that a hit was detected. For a sphere slide, set the parameter between 0 and 2 for the object required.

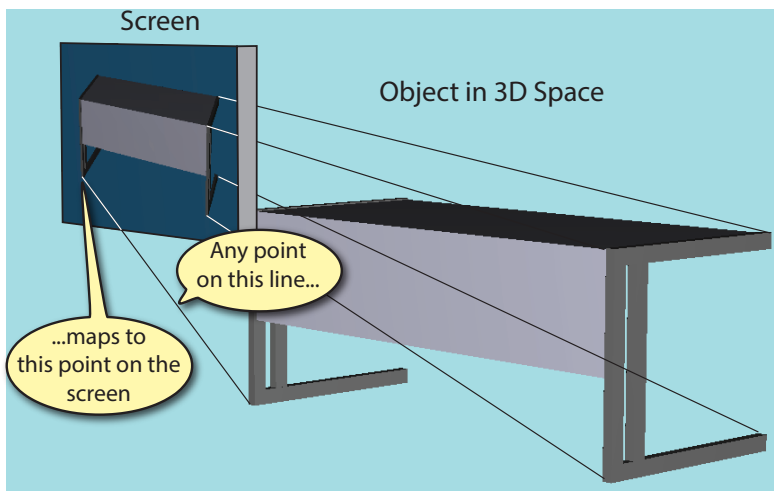
Other 3D Related Statements

Converting Between Screen and 3D Coordinates

It is an everyday thing in our lives to see a three-dimensional world pictured on a two-dimensional screen. The optics of the lens used in the camera to capture this scene do all the required work, focusing the incoming light onto a flat sensor. But when it comes to displaying a virtual 3D world onto a screen, then all the hard work of reducing 3D to 2D is done by mathematics. Any point along a given line will map to a specific point on the screen (see FIG-23.133).

FIG-23.133

How 3D Coordinates Map to the Screen



A change in either the camera's position, where it is pointing, or the zoom factor, all require a recalculation of how the scene appears on the screen.

GetScreenXFrom3D() and GetScreenYFrom3D()

There are times when it is useful to discover where a specified point in 3D space maps to on the screen. To do this, use `GetScreenXFrom3D()` and `GetScreenYFrom3D()` (see FIG-23.134).

FIG-23.134

`GetScreenXFrom3D()`
`GetScreenYFrom3D()`

```
float GetScreenXFrom3D ( x , y , z )  
float GetScreenYFrom3D ( x , y , z )
```

where

x, y, z are real values giving the point in 3D space whose screen coordinates are to be determined.

The values returned by these functions will be in the measurement system being used by your app (virtual pixels or percentage).

The program in FIG-23.135 creates a small sphere centred on the origin. The camera can be moved using a version of the `HandleCamera()` function we created for *FP3D* and the screen coordinates of the centre of the sphere are displayed. This value changes as the camera is moved and hence the screen position of the sphere alters.

FIG-23.135

Displaying 3D Object's
Screen Coordinates

```

rem *** 3D Coords to Screen Coords ***

rem *** Create sphere ***
CreateObjectSphere(1,0.5,15,15)

rem *** Create directional light ***
CreateLightDirectional(1,5,-10,0,255,255,255)

rem *** Position Camera ***
SetCameraPosition(1,0,0,-50)
SetCameraLookAt(1,0,0,-50,0)

rem *** Create text to display results ***
CreateText(1,"")
SetTextSize(1,2)

rem *** Allow camera movement ***
do
    HandleCamera()
    SetTextString(1,"The centre of the sphere maps to screen
    ↵ coords (" + Str(GetScreenXFrom3D(0,0,0),2)+", "+
    ↵ Str(GetScreenYFrom3D(0,0,0),2)+") ")
    Sync()
loop

function HandleCamera()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            rem *** Turn camera ***
            case 37: //left cursor, turn camera left
                RotateCameraGlobalY(1,-1)
            endcase
            case 39: //right cursor, turn camera right
                RotateCameraGlobalY(1,1)
            endcase
            rem *** Move camera options ***
            case 87: //W key, Camera forward ***
                rem *** Make camera parallel to floor ***
                angle = GetCameraAngleX(1)
                RotateCameraLocalX(1,-angle)
                rem *** Move camera ***
                MoveCameraLocalZ(1,0.2)
                rem *** Rem return camera to original rotation ***
                RotateCameraLocalX(1,angle)
            endcase
            case 83: //S key, Camera back ***
                rem *** Make camera parallel to floor ***
                angle = GetCameraAngleX(1)
                RotateCameraLocalX(1,-angle)
                rem *** Move camera ***
                MoveCameraLocalZ(1,-0.2)
                rem *** Rem return camera to original rotation ***
                RotateCameraLocalX(1,angle)
            endcase
            case 38: //up cursor, tilt camera up
                if GetCameraAngleX(1) > -30
                    RotateCameraLocalX(1,-1)
                endif
        endselect
    end if
end function

```



FIG-23.135

(continued)

Displaying 3D Object's
Screen Coordinates

```
        endcase
        case 40: //down cursor,tilt camera down
            if GetCameraAngleX(1) < 30
                RotateCameraLocalX(1,1)
            endif
        endcase
    endselect
endif
endfunction
```

Activity 23.55

Start a new project called *3DToScreen* and implement the code given in FIG-23.135.

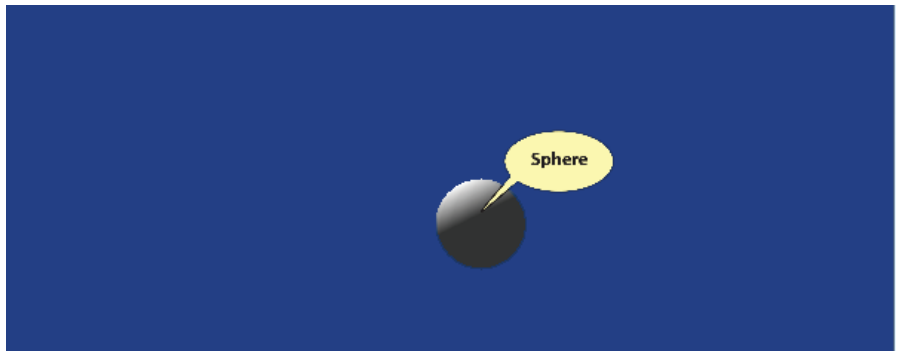
Observe the readings produced as you turn and move the camera.

Save your program.

We can make use of this information to perform tasks such as ensuring that a descriptive sprite is always positioned over a specific 3D position or object. FIG-23.136 shows the use of a sprite to label the sphere in the previous program.

FIG-23.136

Positioning a Sprite
Beside a 3D Object



Remember to copy
BubbleSphere.png to the
project's *media* folder.

Activity 23.56

Adding a sprite label to *3DToScreen* requires the following lines of code to be added at appropriate points in the program:

```
rem *** Load label image ***
LoadImage(1,"BubbleSphere.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,10,-1)
rem *** Sprite offset required so pointer part ***
rem *** of the image is over sphere ***
yoffset = -9
rem *** Position sprite ***
SetSpritePosition(1,GetScreenXFrom3D(0,0,0),
↳GetScreenYFrom3D(0,0,0)+yoffset)
```

Modify *3DToScreen* inserting the lines above at the appropriate points in the code. Test and save your program.

Get3DVectorXFromScreen(), Get3DVectorYFromScreen() and Get3DVectorZFromScreen()

Since we know that all points on a specific line in the 3D world will map to a specific point on the screen, then, of course, the opposite follows: any point on the screen represents a line of points passing through the 3D world.

We can find the offsets of that line for a given point on the screen using the statements `Get3DVectorXFromScreen()`, `Get3DVectorYFromScreen()` and `Get3DVectorZFromScreen()` (see FIG-23.137).

FIG-23.137

`Get3DVectorXFromScreen()`
`Get3DVectorYFromScreen()`
`Get3DVectorZFromScreen()`

float `Get3DVectorXFromScreen` ((`x` , `y`))
float `Get3DVectorYFromScreen` ((`x` , `y`))
float `Get3DVectorZFromScreen` ((`x` , `y`))

where

`x`, `y` are real numbers giving the screen coordinates to which the line maps.

Since everything we see on the screen must pass through the lens of our virtual camera, we know that one end of the line has the same coordinates as that camera.

```
x1 = GetCameraX(1)
y1 = GetCameraY(1)
z1 = GetCameraZ(1)
```

The line's offsets along each axis are given as:

```
xoff# = Get3DVectorXFromScreen(GetPointerX(), GetPointerY())
yoff# = Get3DVectorYFromScreen(GetPointerX(), GetPointerY())
zoff# = Get3DVectorZFromScreen(GetPointerX(), GetPointerY())
```

All we need to do now is multiply the offsets by an appropriate amount so that we create a line of suitable length. For example, if no 3D object on the screen is further than 100 units from the camera, then a multiplier of 100 would be suitable. This value, plus the starting point of the line, will give us its end point.

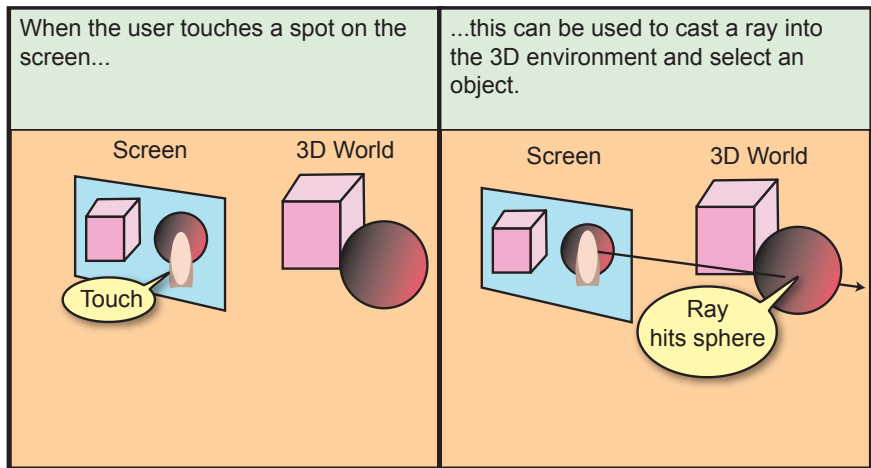
```
x2 = x1 + xoff# * 100
y2 = y1 + yoff# * 100
z2 = z1 + zoff# * 100
```

And now we come to the point of the exercise: if we cast a ray from the start point of the line to the end point, we can detect a hit on any 3D object caused by touching a point on the screen. This allows the user to easily select items within a 3D environment (see FIG-23.138).

In the next program we will use a routine called *HandleTouch()* which will cast a ray from a touched point on the screen into the 3D world. If the ray hits an object, that object will change colour to show that it has been selected. Any previously selected object returns to its original colour when a new one is chosen.

FIG-23.138

Selecting a 3D Object from the Screen



The program is shown in FIG23-139.

FIG-23.139

Implementing Screen Object Selection

```
rem *** Screen Coords to 3D Coords ***

global lastobject = 0  // ID of previous selected object

rem *** Create sphere ***
CreateObjectSphere(1,5,15,15)

rem *** Create cone ***
CreateObjectCone(2,7,3,12)
SetObjectPosition(2,-10,0,0)

rem *** Create box ***
CreateObjectBox(3,6,6,6)
SetObjectPosition(3,10,0,0)

rem *** Create directional light ***
CreateLightDirectional(1,5,-10,0,255,255,255)

rem *** Position Camera ***
SetCameraPosition(1,0,0,-50)
SetCameraLookAt(1,0,0,-50,0)

do
    HandleCamera()
    HandleObject()
    Sync()
loop

function HandleObject()
    rem *** Check for screen press ***
    if GetPointerPressed() = 1
        rem *** Get vector details for point touched ***
        xoff# = Get3DVectorXFromScreen(GetPointerX(),
            ↵GetPointerY())
        yoff# = Get3DVectorYFromScreen(GetPointerX(),
            ↵GetPointerY())
        zoff# = Get3DVectorZFromScreen(GetPointerX(),
            ↵GetPointerY())
        rem *** Get camera's position ***
        x# = GetCameraX(1)
```



FIG-23.139

(continued)

Implementing Screen
Object Selection

```

y# = GetCameraY(1)
z# = GetCameraZ(1)
rem *** Cast a ray from camera with ***
rem *** offset * 100 in all directions ***
hit = ObjectRayCast(0,x#,y#,z#,x#+xoff#*100,y#+yoff#*100,
↳z#+zoff#*100)
rem *** If ray cast hits an object ... ***
if hit <> 0
    rem *** Return previous object to original colour ***
    if lastobject <> 0
        SetObjectColor(lastobject,255,255,255,0)
    endif
    rem *** Set selected object to red tint ***
    SetObjectColor(hit,200,100,100,0)
    rem *** Selected object becomes previous object ***
    lastobject = hit
endif
endif
endfunction

function HandleCamera()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            rem *** Turn camera ***
            case 37: //left cursor, turn camera left
                RotateCameraGlobalY(1,-1)
            endcase
            case 39: //right cursor, turn camera right
                RotateCameraGlobalY(1,1)
            endcase
            rem *** Move camera options ***
            case 87: //W key, Camera forward ***
                rem *** Make camera parallel to floor ***
                angle = GetCameraAngleX(1)
                RotateCameraLocalX(1,-angle)
                rem *** Move camera ***
                MoveCameraLocalZ(1,0.2)
                rem *** Rem return camera to original rotation ***
                RotateCameraLocalX(1,angle)
            endcase
            case 83: //S key, Camera back ***
                rem *** Make camera parallel to floor ***
                angle = GetCameraAngleX(1)
                RotateCameraLocalX(1,-angle)
                rem *** Move camera ***
                MoveCameraLocalZ(1,-0.2)
                rem *** Rem return camera to original rotation ***
                RotateCameraLocalX(1,angle)
            endcase
            case 38: //up cursor,tilt camera up
                if GetCameraAngleX(1) > -30
                    RotateCameraLocalX(1,-1)
                endif
            endcase
            case 40: //down cursor,tilt camera down
                if GetCameraAngleX(1) < 30
                    RotateCameraLocalX(1,1)
                endif
            endcase
        endselect
    end if
endfunction

```



FIG-23.139

(continued)

Implementing Screen
Object Selection

```

                endcase
            endselect
        endif
    endfunction

```

Activity 23.57

Start a new project called *ScreenTo3D* and implement the code given in FIG-23.139. Check that the 3D objects can be selected correctly irrespective of the camera's position. Save your program.

Sprite and 3D Objects Depth Settings

SetGlobal3DDepth()

When you mix 3D objects and sprites, you can define which sprite depth all 3D objects are to be drawn on. By default, 3D objects are placed on layer 5,000 and sprites on layer 10, so sprites always appear in front of 3D objects. Although you cannot position individual 3D objects on varying layers, you can specify which layer all 3D objects should be drawn on using the statement `SetGlobal3DDepth()` (see FIG-23.140).

FIG-23.140

SetGlobal3DDepth()

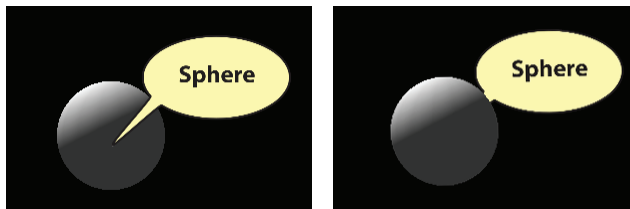
The diagram shows the function call `SetGlobal3DDepth()` with its parameters. `SetGlobal3DDepth` is in an orange box, followed by an opening parenthesis `(` in a light blue box, then the parameter `idepth` in a green box, and finally a closing parenthesis `)` in a light blue box.

where

idepth

is an integer value (0 to 10,000) giving the depth at which all 3D objects are to be drawn (default is 5,000).

FIG-23.141 shows the result of mixing sprites and 3D objects at differing depths.

FIG-23.141The Effects of Changing
3D Depth

Default:
Sphere on layer 5,000
Sprite on layer 10

Sphere on layer 5
Sprite on layer 10

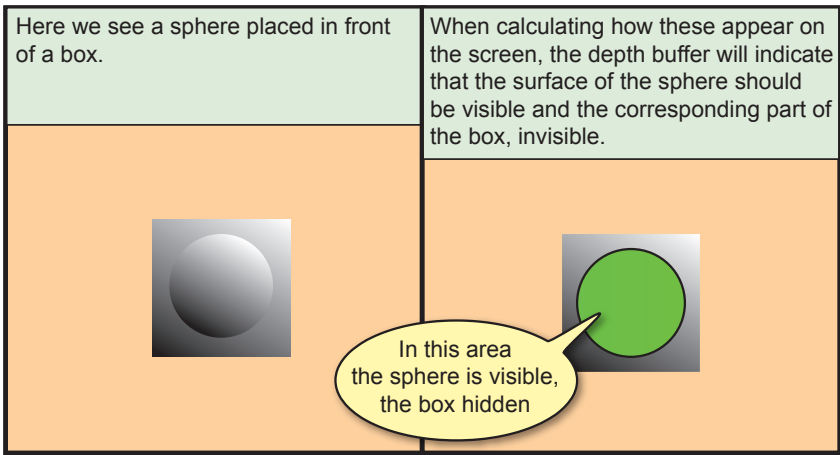
When sprites and 3D objects are on the same layer, sprites always appear in front of the 3D objects.

The Depth Buffer

In order to help with the calculations of mapping the 3D virtual world onto the screen, AGK maintains a depth buffer. This buffer contains information on the position of objects relative to each other from the current camera position. By knowing which object appears behind another object for any given point on the screen, AGK can determine which parts of an object should be drawn and which should be ignored (see FIG-23.142).

FIG-23.142

Deciding On How an Object is Displayed



However, it is not always the case that the front object should obscure the one behind it. For example, if the sphere was transparent, then the surface of the box should show through.

For each object, we can set a test condition which defines how the depth information in the depth buffer will determine if that object should appear at any given point on the screen.

SetObjectDepthReadMode()

To change the test performed when determining if an object's surface should be visible when AGK constructs the screen layout of a 3D screen, use the `SetObjectDepthReadMode()` statement (see FIG-23.143).

FIG-23.143

`SetObjectDepthReadMode(` **SetObjectDepthReadMode** `(` **id** `,` **imode** `)`

where

id is an integer value giving the ID of the object whose test condition is to be changed.

imode is an integer value (0 to 8) which indicates the test to be performed. The default value is 1.

FIG-23.144

Mode Options

FIG-23.144 explains the meaning of each *imode* value.

imode	Meaning	Result	imode	Meaning	Result
0	never	The surface will never show	4	>	Shows when depth greater than others
1	<	Shows when depth less than others	5	≠	Shows when depth not equal to others
2	=	Shows when depth equal to others	6	>=	Shows when depth greater than or equal to others
3	<=	Shows when depth less than or equal to others	7	always	The surface will always show

The program in FIG-23.145 demonstrates the effect of each test on an intersecting cube and sphere.

FIG-23.145

The Effects of Change
Read Mode

```
rem *** Depth Read Mode ***
global dmode = 1

rem *** Create sphere ***
CreateObjectSphere(1,5,15,15)

rem *** Create box ***
CreateObjectBox(3,6,6,6)
SetObjectPosition(3,0,0,2)

rem *** Create directional light ***
CreateLightDirectional(1,5,-10,0,255,255,255)

rem *** Position Camera ***
SetCameraPosition(1,-13,0,-10)
SetCameraLookAt(1,0,0,0,0)

rem *** Text to show current mode ***
CreateText(1,"")

do
    ChangeDepthTest()
    HandleCamera()
    SetTextString(1,"Depth mode: "+Str(dmode))
    Sync()
loop

function ChangeDepthTest()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyPressed(key)=1
        select key
            rem *** Turn camera ***
            case 13: //Enter key, inc mode
                dmode = (dmode + 1) mod 8
                SetObjectDepthReadMode(3,dmode)
            endcase
        endselect
    endif
endfunction
```

Activity 23.58

Start a new project called *DepthMode3D* and implement the code given in FIG-23.145. Observe the results obtained in each mode. Save your program.

SetObjectDepthWrite()

It is possible to stop an object writing information to the depth buffer. In this situation, all other objects will be assumed to appear in front of such an object.

You can turn stop an object recording its details in the depth buffer using the `SetObjectDepthWrite()` statement (see FIG-23.146).

FIG-23.146

SetObjectDepthWrite()

 The diagram shows the function signature `SetObjectDepthWrite()` with a yellow highlight. The parameters are `(id , imode)`, where `id` and `imode` are highlighted in green.

where

id is an integer value giving the ID of the object whose write mode is to be set.

imode is an integer value (0 or 1). Using zero will mean that object's details are not written to the depth buffer; 1 writes depth information as normal.

A sky box is a large cube which has been textured with an image of the sky on its inside surfaces. All other objects are placed within the box to give the impression of being surrounded by sky.

The most likely scenario for using this option is when your program makes use of a sky box. By drawing the box with *depth-write* off, no information about that object is recorded in the depth buffer. When other 3D objects are added, they will see an empty depth buffer and assume they are free to draw themselves to the back buffer. Regardless of where the sky box is in world space, it will be overwritten by all other objects in the back buffer.

GetObjectDepthReadMode()

To discover the current read depth test condition being used for a given object, use the statement `GetObjectDepthReadMode()` (see FIG-23.147).

FIG-23.147

GetObjectDepthReadMode()

 The diagram shows the function signature `GetObjectDepthReadMode()` with a yellow highlight. The parameter is `(id)`, where `id` is highlighted in green.

where

id is an integer value giving the ID of the object whose depth read mode is to be found.

The function will return a value between 0 and 7.

GetObjectDepthWrite()

To discover if an object writes to the depth buffer, use the statement `GetObjectDepthWrite()` (see FIG-23.148).

FIG-23.148

GetObjectDepthWrite()

 The diagram shows the function signature `GetObjectDepthWrite()` with a yellow highlight. The parameter is `(id)`, where `id` is highlighted in green.

where

id is an integer value giving the ID of the object whose depth read mode is to be found.

The function will return the value 0 for objects that do not write to the depth buffer; 1 for all other objects.

Shaders

A shader is a program designed to be run by a machine's graphics processing unit (GPU) and is used to affect the lighting of a scene or to create special effects.

It is beyond the scope of this text to discuss the creation of a new shader, but AGK does contain commands for loading an existing shader program, setting shader

program variables and specifying which shader is to be used on specific objects in a scene.

LoadShader()

To load a new shader program, use the `LoadShader()` statement (see FIG-23.149).

FIG-23.149

LoadShader()

Format 1

`LoadShader (id , vfile , pfile)`

Format 2

`integer LoadShader (id , vfile , pfile)`

where

id	is an integer value giving the ID to be assigned to the new shader.
vfile	is a string value giving the name of the file containing the vertex shader. Normally, these files have a <code>.vs</code> extension.
pfile	is a string value giving the name of the file containing the pixel shader. Normally, these files have a <code>.ps</code> extension.

The vertex shader transforms vertex data into screen position data. This allows the original shape within the model to be reshaped when it appears on the screen.

The pixel shader modifies the colour of each screen pixel.

SetObjectShader()

To assign a loaded shader to a specific object, use `SetObjectShader()` (see FIG-23.150).

FIG-23.150

SetObjectShader()

`SetObjectShader (id , idshd)`

where

id	is an integer value giving the ID of an existing object.
idshd	is an integer value giving the ID of the shader to be used.

AGK has its own default shader which has an ID value of zero.

The program in FIG-23.151 loads a simple shader and applies it to a box object.

FIG-23.151

Using a Shader

```
rem *** Using a Shader ***  
  
rem *** Create Box ***  
CreateObjectBox(1,6,6,6)  
rem *** Create directional light ***  
CreateLightDirectional(1,5,-10,0,255,255,255)
```



FIG-23.151

(continued)

Using a Shader

Thanks to Paul Johnston for supplying the shader files.

```
LoadImage(1,"Wood.png")
SetObjectImage(1,1,0)

rem *** Position Camera ***
SetCameraPosition(1,-13,0,-10)
SetCameraLookAt(1,0,0,0,0)

rem *** Load shader ***
LoadShader(1,"vertex.vs","pixel.ps")

rem *** Assign shader to box ***
SetObjectShader(1,1)

rem *** display result ***
do
    Sync()
loop
```

Activity 23.59

Start a new project called *Shader3D* and implement the code given in FIG-23.151. Copy the files *vertex.vs* and *pixel.ps* into the project's *media* folder.

Run the program twice. On the second run, comment out the `SetObjectShader()` statement to see how this affects the result.

Save your program.

FIG-23.152

SetShaderConstantBy
Name()

SetShaderConstantByName()

Named constants used within a shader file can be modified using the `SetShaderConstantByName()` statement (see FIG-23.152).

```
SetShaderConstantByName ( ( id , name , v1 , v2 , v3 , v4 )
```

where

id	is an integer value giving the ID of the loaded shader containing the constant.
name	is a string value giving the name of the constant.
v1, v2, v3, v4	are real numbers giving the value to be assigned to the constant.

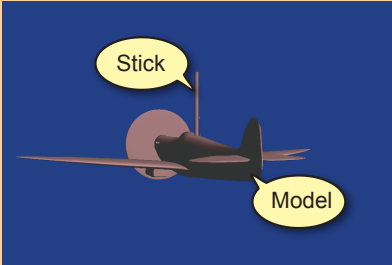
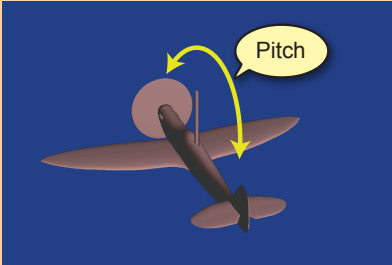
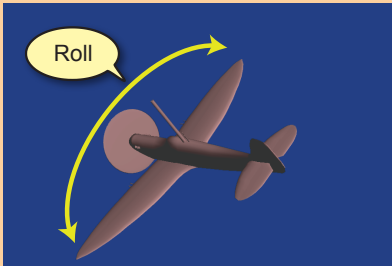

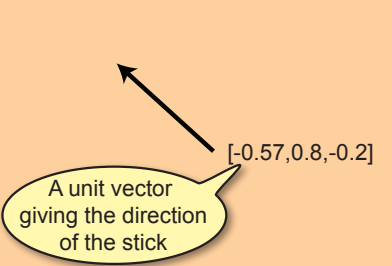
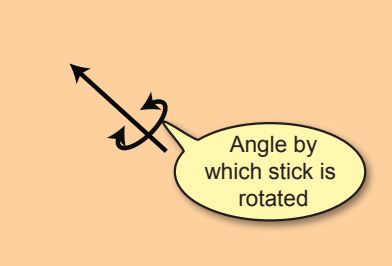
As you can see, there are four value parameters. How many of these are needed depends on the type of constant whose value is being specified. For example, if the constant is an integer or real one, then only the first value parameter (*v1*) will be used. However, if the constant represents a 3D vector, then the first three value parameters (*v1*, *v2*, and *v3*) will be used.

Note that you must always specify amounts for each of the four value parameters (*v1* to *v4*) when calling the function. Only those required by the shader constant will be used.

Quaternion Rotation

Under most circumstances the easiest way of rotating an object is to define its position in terms of angles of rotation about the object's local x, y and z-axes. And this is exactly how the `RotateObjectLocalX()`, `RotateObjectLocalY()`, and `RotateObjectLocalZ()` statements operate. However, there are a few rare occasions when defining an object's orientation in this way can give rise to ambiguous or unwanted results. To avoid these problems, the orientation of any object can be set using a **quaternion** value. A quaternion value is a four element vector traditionally written as $[w,x,y,z]$. FIG-23.153 helps demonstrate what a quaternion value represents.

FIG-23.153
Defining the Orientation
of an Object

Imagine a clay model version of an object which has a cocktail stick embedded.	If we treat the stick in the same manner as a game joystick, pushing it forward or back will add pitch...
	
...pushing to the side will add roll.	Finally, rotating the stick gives yaw.
	
If we convert our real-life model into a more mathematical one, we can represent the direction in which the stick is pointing as a unit vector...	...and its rotation as a single angle.
	

You should have realised from the demonstration shown in FIG-23.153 that our clay model can be placed in any orientation by a combination of pointing the cocktail stick in a specific direction and then rotating the model about the stick by an appropriate angle.

Looking at the operation in reverse, a model's orientation can be specified by a vector equivalent in direction to the positive half of the model's local y-axis and the angle of rotation about that vector.

The four numbers in a quaternion are directly related to the angle and vector shown in FIG-23.153. Let's assume that a 3D object has been positioned in such a way as to create a unit vector $[x,y,z]$ and that it has been rotated by θ radians about that vector, the quaternion equivalent of this would be

$$[\cos \theta/2, x\sin(\theta/2), y\sin(\theta/2), z\sin(\theta/2)]$$

Both objects and cameras can be orientated using quaternion values.

SetObjectRotationQuat()

To orientate a 3D object using a quaternion value, use `SetObjectRotationQuat()` (see FIG-23.154).

FIG-23.154

`SetObjectRotationQuat()`

`SetObjectRotationQuat ((id , w , x , y , z)`

where

id is an integer value giving the ID of the object.

w, x, y, z are real values giving the quaternion.

SetCameraRotationQuat()

To orientate a camera using a quaternion value, use `SetCameraRotationQuat()` (see FIG-23.155).

FIG-23.155

`SetCameraRotationQuat()`

`SetCameraRotationQuat ((id , w , x , y , z)`

where

id is an integer value giving the ID of the camera.

w, x, y, z are real values giving the quaternion.

GetObjectQuatW(), GetObjectQuatX(), GetObjectQuatY() and GetObjectQuatZ()

To find the quaternion representing the current orientation of a 3D object, use `GetObjectQuatW()`, `GetObjectQuatX()`, `GetObjectQuatY()` and `GetObjectQuatZ()` (see FIG-23.156).

FIG-23.156

GetObjectQuatW()
 GetObjectQuatX()
 GetObjectQuatY()
 GetObjectQuatZ()

```
float GetObjectQuatW ( ( id ) )
float GetObjectQuatX ( ( id ) )
float GetObjectQuatY ( ( id ) )
float GetObjectQuatZ ( ( id ) )
```

where

id is an integer value giving the ID of the object.

GetCameraQuatW(), GetCameraQuatX(), GetCameraQuatY(), and GetCameraQuatZ()

To find the quaternion representing the current orientation of a camera, use `GetCameraQuatW()`, `GetCameraQuatX()`, `GetCameraQuatY()` and `GetCameraQuatZ()` (see FIG-23.157).

FIG-23.157

GetCameraQuatW()
 GetCameraQuatX()
 GetCameraQuatY()
 GetCameraQuatZ()

```
float GetCameraQuatW ( ( id ) )
float GetCameraQuatX ( ( id ) )
float GetCameraQuatY ( ( id ) )
float GetCameraQuatZ ( ( id ) )
```

where

id is an integer value giving the ID of the camera.

Summary

- Use `GetScreenXFrom3D()` and `GetScreenYFrom3D()` to discover what position on the screen a point in 3D space maps to.
- Use `Get3DVectorXFromScreen()`, `Get3DVectorYFromScreen()` and `Get3DVectorZFromScreen()` in combination with the camera's position to discover the line in 3D space which maps to a given screen position.
- Use `SetGlobal3DDepth()` to specify which sprite layer 3D objects are to be placed on.
- AGK's depth buffer is used to help determine the depth of objects relative to the current camera's position.
- Use `SetObjectDepthReadMode()` to change the condition used to determine when a surface element is written to the screen's back buffer.
- Use `SetObjectDepthWrite()` to control an object's details being written to the depth buffer.
- Use `GetObjectDepthReadMode()` to determine an object's current test

condition for being written to the screen.

- Use `GetObjectDepthWrite()` to determine if a given object writes its details to the depth buffer.
- Use `LoadShader()` to create a new shader.
- An AGK shader is constructed from two separate files - a vertex shader and a pixel shader.
- Use `SetObjectShader()` to apply a specific shader to an object.
- Use `SetShaderConstantByName()` to assign a value to a named shader constant.
- A quaternion is a sequence of four real values usually written as $[w, x, y, z]$.
- A quaternion value can be used to specify an axis vector and an angle of rotation about that axis.
- Use `SetObjectRoatationQuat()` to specify the orientation of a 3D object using a quaternion value.
- Use `SetCameraRotationQuat()` to specify the orientation of a camera using a quaternion value.
- Use `GetObjectQuatW()`, `GetObjectQuatX()`, `GetObjectQuatY()` and `GetObjectQuatZ()` to obtain the quaternion values of a given object.
- Use `GetCameraQuatW()`, `GetCameraQuatX()`, `GetCameraQuatY()` and `GetCameraQuatZ()` to obtain the quaternion values of a given camera.

Solutions

Activity 23.1

```
length = Sqrt(42 + 32 + 72)
        = Sqrt(16 + 9 + 49)
        = Sqrt(74)
        = 8.6 (approx)
```

Activity 23.2

No solution required.

Activity 23.3

Modified code for *First3D* (first change):

```
rem *** Position Camera ***

rem *** Load model ***
LoadObject(1,"Axes.obj",15)

rem *** Position camera ***
SetCameraPosition(1,20,10,-20)

rem *** Display model ***
do
    Sync()
loop
```

No part of the axes model is visible.

Modified code for *First3D* (second change):

```
rem *** Moving Camera ***

rem *** Load model ***
LoadObject(1,"Axes.obj",15)

rem *** Camera's position on x-axis ***
x# = 0

rem *** Move camera until 20 units along x-axis ***
repeat
    rem *** Reposition camera ***
    x# = x# + 0.1
    SetCameraPosition(1,x#,10,-20)
    Sync()
    rem *** Wait 50 msecs ***
    Sleep(50)
until x# >= 20

rem *** Do nothing ***
do
    Sync()
loop
```

Activity 23.4

Modified code for *First3D*:

```
rem *** Adjusting Camera Aim ***
rem *** Load model ***
LoadObject(1,"Axes.obj",15)
rem *** Camera's position on x-axis ***
x# = 0
rem *** Move camera until 20 units along x-axis ***
repeat
    rem *** Reposition camera ***
    x# = x# + 0.1
    SetCameraPosition(1,x#,10,-20)
    rem *** Point camera at origin ***
    SetCameraLookAt(1,0,0,0,0)
    rem *** Update screen ***
    Sync()
    rem *** Wait 50 msecs ***
    Sleep(50)
until x# >= 20
rem *** Do nothing ***
do
    Sync()
loop
```

Activity 23.5

Modified code for *First3D* (red light used):

```
rem *** Adjusting Camera Aim ***
rem *** Load model ***
LoadObject(1,"Axes.obj",15)
rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,200,150,150)
rem *** Camera's position on x-axis ***
x# = 0
rem *** Move camera until 20 units along x-axis ***
repeat
    rem *** Reposition camera ***
    x# = x# + 0.1
    SetCameraPosition(1,x#,10,-20)
    rem *** Point camera at origin ***
    SetCameraLookAt(1,0,0,0,0)
    rem *** Update screen ***
    Sync()
    rem *** Wait 50 msecs ***
    Sleep(50)
until x# >= 20
rem *** Do nothing ***
do
    Sync()
loop
```

Activity 23.6

No solution required.

Activity 23.7

Modified code for *Second3D*:

```
rem ***User Controlled Camera Movement with Roll ***

rem *** Structure for camera details ***
type CameraDataType
    x as float
    y as float
    z as float // Camera's coords
    dist as float // Camera's distance from y-axis
    angle as float // Camera's rotation about y-axis
    roll as float // Camera's roll
endtype

rem *** Global Variable ***
rem *** Camera info ***
global camera as CameraDataType
camera.x = 0
camera.y = 10
camera.z = -25
camera.dist = 25
camera.angle = -90
camera.roll = 0

rem *** Load model ***
LoadObject(1,"Axes.obj",20)

rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,200,150,150)

rem *** Add text to show camera's coords ***
CreateText(1,"")
CreateText(2,"")
SetTextPosition(2,0,6)
CreateText(3,"")
SetTextPosition(3,0,12)

rem *** Give user camera control ***
do
    HandleCamera()
    rem *** Show camera position ***
    SetTextString(1,"X: "+Str(camera.x,1))
    SetTextString(2,"Y: "+Str(camera.y,1))
    SetTextString(3,"Z: "+Str(camera.z,1))
    rem *** Update screen ***
    Sync()
loop

function HandleCamera()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
```

```

if GetRawKeyState(key)=1
  select key
    case 37: //left cursor, decrease angle
      dec camera.angle
    endcase
    case 38: //up cursor, increase y
      camera.y=camera.y+0.25
    endcase
    case 39: //right cursor, increase angle
      inc camera.angle
    endcase
    case 40: //down cursor, decrease y
      camera.y=camera.y-0.25
    endcase
    case 187: //+ key roll +1
      camera.roll = camera.roll + 1
    endcase
    case 189: //- key; roll -1
      camera.roll = camera.roll - 1
    endcase
  endselect
endif
rem *** Calculate new x and z coordinates ***
camera.x = camera.dist*cos(camera.angle)
camera.z = camera.dist*sin(camera.angle)
rem *** Reposition camera to match ***
SetCameraPosition(1,camera.x,camera.y,camera.z)
rem *** Make camera point at origin ***
SetCameraLookAt(1,0,0,camera.roll)
endfunction

```

Activity 23.8

The new statements required when creating the box are shown below in context with existing statements:

```

rem *** Add text to show camera's coords ***
CreateText(1,"")
CreateText(2,"")
SetTextPosition(2,0,6)
CreateText(3,"")
SetTextPosition(3,0,12)

rem *** Add box to model ***
CreateObjectBox(2,3,2,1)
rem *** Give user camera control ***
do

```

The box's centre as at the origin.

Activity 23.9

To create a sphere, change the lines

```

rem *** Add box to model ***
CreateObjectBox(2,3,2,1)

```

to

```

rem *** Add sphere to model ***
CreateObjectSphere(2,5,4,4)

```

The shape created in this version of the statement is a poor approximation of a sphere.

In the second version the sphere is created using the statement

```

CreateObjectSphere(2,5,20,20)

```

This produces a realistic sphere.

Activity 23.10

To create a cone, change the lines

```

rem *** Add sphere to model ***
CreateObjectSphere(2,5,20,20)

```

to

```

rem *** Add cone to model ***
CreateObjectCone(2,6,2.5,5)

```

The shape created in this version of the statement is a poor approximation of a cone.

In the second version the cone is created using the statement

```

CreateObjectCone(2,6,2.5,15)

```

This produces a realistic cone.

Activity 23.11

To create a cylinder, change the lines

```

rem *** Add cone to model ***
CreateObjectCone(2,6,2.5,15)

```

to

```

rem *** Add cylinder to model ***
CreateObjectCylinder(2,4,3,12)

```

Activity 23.12

To add a plane to the existing model, include the lines

```

rem *** Add plane ***
CreateObjectPlane(3,8,7)

```

Visually, the plane cuts the cylinder in two.

Unlike standard polygons, a plane can be viewed from both sides.

Activity 23.13

No solution required.

Activity 23.14

The function required to delete the cylinder contains the following code:

```

function HandleCylinderDelete()
  rem *** Get last key pressed ***
  key = GetRawLastKey()
  rem *** If key currently pressed, process it***
  if GetRawKeyState(key)=1
    select key
      case 46: //delete key, delete cylinder
        if GetObjectExists(3)=1
          DeleteObject(3)
        endif
      endcase
    endselect
  endif
endfunction

```

Although using a `select` structure may seem like a bit of overkill for a single option, it will make it easier to add any future options that might be required.

The function must be called within the `do..loop` structure of the main section of the program with the line

```

HandleCylinderDelete()

```

Although the positioning of the call is not critical, it is perhaps best to place it immediately after the existing function call to `HandleCamera()`.

Activity 23.15

The colour of the directional light is changed by updating the statement creating that light to read:

```

rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,250,250,250)

```

The plane is created and coloured using the lines:

```

rem *** Add plane ***
CreateObjectPlane(3,8,7)
rem *** Set colour of plane ***
SetObjectColor(3,90,150,200,255)

```


Activity 23.16

The latest version of the main section of *Second3D* is coded as:

```
rem *** Texturing an Object ***

rem *** Structure for camera details ***
type CameraDataType
  x as float
  y as float
  z as float      // Camera's coords
  dist as float   // Camera's distance from y-axis
  angle as float  // Camera's rotation about y-axis
  roll as float   // Camer's roll
endtype

rem *** Global Variable ***
rem *** Camera info ***
global camera as CameraDataType
camera.x = 0
camera.y = 10
camera.z = -25
camera.dist = 25
camera.angle = -90
camera.roll = 0

rem *** Load model ***
LoadObject(1,"Axes.obj",20)

rem *** Add box ***
CreateObjectBox(2,6,6,6)
rem *** Texture box ***
LoadImage(1,"Wood.png")
SetObjectImage(2,1,0)

rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,250,250,250)

rem *** Add text to show camera's coords ***
CreateText(1,"")
CreateText(2,"")
SetTextPosition(2,0,6)
CreateText(3,"")
SetTextPosition(3,0,12)

rem *** Give user camera control ***
do
  HandleCamera()
  rem *** Show camera position ***
  SetTextString(1,"X: "+Str(camera.x,1))
  SetTextString(2,"Y: "+Str(camera.y,1))
  SetTextString(3,"Z: "+Str(camera.z,1))
  rem *** Update screen ***
  Sync()
loop
```

The code for *HandleCamera()* is unchanged.

Activity 23.17

No solution required.

Activity 23.18

In *Second3D*, replace the lines

```
rem *** Load image ***
LoadImage(1,"Wood.png")
rem *** Assign image as texture of box ***
SetObjectImage(2,1,0)
```

with

```
rem *** Load image ***
LoadImage(1,"Perimeter.png")
rem *** Assign image as texture of box ***
SetObjectImage(2,1,0)
rem *** Activate transparency ***
SetObjectTransparency(2,1)
```

Activity 23.19

To display only front faces, change the *SetObjectCullMode()* statement to read:

```
SetObjectCullMode(2,1)
```

Only the front section of the hemisphere is visible. That is, the part of the model displaying front faces.

To display only back faces, change the *SetObjectCullMode()* statement to read:

```
SetObjectCullMode(2,2)
```

With the front facing polygon omitted, a larger number of back facing polygons become visible.

Activity 23.20

If the base of the box is to sit on the XZ plane, it needs to be raised by 3 units in the y direction.

After the box is created, move it to its new position by adding the lines

```
rem *** Position box ***
SetObjectPosition(2,0,3,0)
```

This places the centre of the box at (0,3,0) and hence the bottom of the box will be at zero on the y-axis.

Activity 23.21

The code to the *HandleBox()* function in *Second3D*:

```
function HandleBox()
  rem *** Get last key pressed ***
  key = GetRawLastKey()
  rem *** If key currently pressed, process it***
  if GetRawKeyState(key)=1
    select key
      rem *** Move object options ***
      case 82: //R key, object right
        MoveObjectLocalX(2,0.1)
      endcase
      case 76: //L key, object left
        MoveObjectLocalX(2,-0.1)
      endcase
      case 85: //U key, object up
        MoveObjectLocalY(2,0.1)
      endcase
      case 68: //D key, object down
        MoveObjectLocalY(2,-0.1)
      endcase
      case 73: //I key, object in
        MoveObjectLocalZ(2,0.1)
      endcase
      case 79: //O key, object out
        MoveObjectLocalZ(2,-0.1)
      endcase
    endselect
  endif
endfunction
```

A call to this function should be placed after the call to *HandleCamera()* in the main section of the program.

Activity 23.22

After creating the box, add the statements

```
rem *** Rotate box 20 degrees about z-axis ***
SetObjectRotation(2,0,0,20)
```

Since the local x and y axes have now been rotated, the direction of the box's movement also changes with respect to the viewer (except along the local z-axis, whose orientation is unchanged).

Activity 23.23

To allow rotation about the box's local axes, modify *HandleBox()*'s code to be:

```
function HandleBox()
  rem *** Get last key pressed ***
  key = GetRawLastKey()
  rem *** If key currently pressed, process it***
```

```

if GetRawKeyState(key)=1
select key
rem *** Move object options ***
case 82: //R key, object right
MoveObjectLocalX(2,0.1)
endcase
case 76: //L key, object left
MoveObjectLocalX(2,-0.1)
endcase
case 85: //U key, object up
MoveObjectLocalY(2,0.1)
endcase
case 68: //D key, object down
MoveObjectLocalY(2,-0.1)
endcase
case 73: //I key, object in
MoveObjectLocalZ(2,0.1)
endcase
case 79: //O key, object out
MoveObjectLocalZ(2,-0.1)
endcase
rem *** Rotate Object about local axes ***
case 81: //Q key, x-axis + 1
RotateObjectLocalX(2,1)
endcase
case 87: //W key, x-axis - 1
RotateObjectLocalX(2,-1)
endcase
case 69: //E key, y-axis + 1
RotateObjectLocalY(2,1)
endcase
case 67: //C key, y-axis - 1
RotateObjectLocalY(2,-1)
endcase
case 75: //K key, z-axis + 1
RotateObjectLocalZ(2,1)
endcase
case 77: //M key, z-axis - 1
RotateObjectLocalZ(2,-1)
endcase
endselect
endif
endfunction

```

Activity 23.24

The code for the final version of *Second3D*:

```

rem *** Texturing an Object ***

rem *** Structure for camera details ***
type CameraDataType
x as float
y as float
z as float // Camera's coords
dist as float // Camera's distance from y-axis
angle as float // Camera's rotation about y-axis
roll as float // Camer's roll
endtype

rem *** Global Variable ***
rem *** Camera info ***
global camera as CameraDataType
camera.x = 0
camera.y = 10
camera.z = -25
camera.dist = 25
camera.angle = -90
camera.roll = 0

rem *** Load model ***
LoadObject(1,"Axes.obj",20)

rem *** Add box ***
CreateObjectBox(2,6,6,6)
rem *** Position box ***
SetObjectPosition(2,0,3,0)
rem *** Texture box ***
LoadImage(1,"Perimeter.png")
SetObjectImage(2,1,0)
SetObjectTransparency(2,1)

rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,250,250,250)

rem *** Add text to show camera's coords ***
CreateText(1,"")
CreateText(2,"")
SetTextPosition(2,0,6)

```

```

CreateText(3,"")
SetTextPosition(3,0,12)

rem *** Give user camera control ***
do
HandleCamera()
HandleBox()
rem *** Show camera position ***
SetTextString(1,"X: "+Str(camera.x,1))
SetTextString(2,"Y: "+Str(camera.y,1))
SetTextString(3,"Z: "+Str(camera.z,1))
rem *** Update screen ***
Sync()
loop

function HandleCamera()
rem *** Get last key pressed ***
key = GetRawLastKey()
rem *** If key currently pressed, process it***
if GetRawKeyState(key)=1
select key
case 37: //left cursor, decrease angle
dec camera.angle
endcase
case 38: //up cursor, increase y
camera.y=camera.y+0.25
endcase
case 39: //right cursor, increase angle
inc camera.angle
endcase
case 40: //down cursor, decrease y
camera.y=camera.y-0.25
endcase
case 187: //+ key roll +1
camera.roll = camera.roll + 1
endcase
case 189: //- key; roll -1
camera.roll = camera.roll - 1
endcase
endselect
endif
rem *** Calculate new x and z coordinates ***
camera.x = camera.dist*cos(camera.angle)
camera.z = camera.dist*sin(camera.angle)
rem *** Reposition camera to match ***
SetCameraPosition(1,camera.x,camera.y,camera.z)
rem *** Make camera point at origin ***
SetCameraLookAt(1,0,0,0,camera.roll)
endfunction

function HandleBox()
rem *** Get last key pressed ***
key = GetRawLastKey()
rem *** If key currently pressed, process it***
if GetRawKeyState(key)=1
select key
rem *** Move object options ***
case 82: //R key, object right
MoveObjectLocalX(2,0.1)
endcase
case 76: //L key, object left
MoveObjectLocalX(2,-0.1)
endcase
case 85: //U key, object up
MoveObjectLocalY(2,0.1)
endcase
case 68: //D key, object down
MoveObjectLocalY(2,-0.1)
endcase
case 73: //I key, object in
MoveObjectLocalZ(2,0.1)
endcase
case 79: //O key, object out
MoveObjectLocalZ(2,-0.1)
endcase
rem *** Rotate Object about local axes ***
case 81: //Q key, x-axis + 1
RotateObjectLocalX(2,1)
endcase
case 87: //W key, x-axis - 1
RotateObjectLocalX(2,-1)
endcase
case 69: //E key, y-axis + 1
RotateObjectLocalY(2,1)
endcase
case 67: //C key, y-axis - 1
RotateObjectLocalY(2,-1)
endcase

```

```

case 75: //K key, z-axis + 1
    RotateObjectLocalZ(2,1)
endcase
case 77: //M key, z-axis - 1
    RotateObjectLocalZ(2,-1)
endcase
rem *** Rotate object about inertial axes ***
case 90: //Z key, x-axis + 1
    RotateObjectGlobalX(2,1)
endcase
case 88: //X key, x-axis - 1
    RotateObjectGlobalX(2,-1)
endcase
case 70: //F key, y-axis + 1
    RotateObjectGlobalY(2,1)
endcase
case 86: //V key, y-axis - 1
    RotateObjectGlobalY(2,-1)
endcase
case 66: //B key, z-axis + 1
    RotateObjectGlobalZ(2,1)
endcase
case 78: //N key, z-axis - 1
    RotateObjectGlobalZ(2,-1)
endcase
endsselect
endif
endfunction

```

Activity 23.25

No solution required.

Activity 23.26

To make the shape expand in the z direction rather than the x direction, change the `SetObjectScale()` statement to read

```
SetObjectScale(1,1,1,scale#)
```

Activity 23.27

Modified code for *Zoom3D*:

```

rem *** FOV Demo ***

rem *** Load Model ***
LoadObject(1,"Robot.obj",15)

rem *** Apply texture ***
LoadImage(1,"Robotskin.png")
SetObjectImage(1,1,0)

rem *** Create background plane ***
CreateObjectPlane (2,100,100)
SetObjectPosition(2,0,0,10)
LoadImage(2,"Background.jpg")
SetObjectImage(2,2,0)

rem *** Create Directional light ***
CreateLightDirectional(1,10,10,10,255,255,255)

rem *** Position and point camera ***
SetCameraPosition(1,0,10,-30)
SetCameraLookAt(1,0,5,0,0)

rem *** set field of view ***
global fov = 90

do
    HandleZoom()
    Sync()
loop

function HandleZoom()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            case 33: //Page Up, zoom in
                if fov > 5
                    dec fov
                endif
            endcase
            case 34: //Page Down, zoom out
                if fov < 100

```

```

inc fov
endif
endcase
endselect
endif
SetCameraFOV(1,fov)
endfunction

```

Note that the variable *fov* has become a global variable. Alternatively, it could have become a IN/OUT parameter to the function.

Activity 23.28

To display both front and back faces of the three objects, add the lines

```

rem *** Show front and back faces of all objects ***
SetObjectCullMode(1,0)
SetObjectCullMode(2,0)
SetObjectCullMode(3,0)

```

after creating all three objects.

Activity 23.29

No solution required.

Activity 23.30

The modified version of *FP3D*:

```

rem *** First Person Perspective ***

rem *** Load textured walls ***
LoadObject(1,"Maze.obj",20)
LoadImage(1,"BricksLarge.png")
SetObjectImage(1,1,0)
rem *** Load floor ***
CreateObjectPlane(2,180,180)
RotateObjectLocalX(2,90)
SetObjectPosition(2,0,1,0)
SetObjectColor(2,220,180,180,0)

rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,250,250,250)

rem *** Position camera ***
SetCameraPosition(1,0,5,-10)
SetCameraLookAt(1,0,5,-10,0)
rem *** Set camera's field of view to 80 ***
SetCameraFOV(1,80)

rem *** View scene ***
do
    Sync()
loop

```

The visible area becomes larger when the field of view is changed.

Activity 23.31

Modified code for *FP3D*:

```

rem *** First Person Perspective ***

rem *** Load textured walls ***
LoadObject(1,"Maze.obj",20)
LoadImage(1,"BricksLarge.png")
SetObjectImage(1,1,0)
rem *** Load floor ***
CreateObjectPlane(2,180,180)
RotateObjectLocalX(2,90)
SetObjectPosition(2,0,1,0)
SetObjectColor(2,220,180,180,0)

rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,250,250,250)

rem *** Position camera ***
SetCameraPosition(1,0,5,-10)
SetCameraLookAt(1,0,5,-10,0)
rem *** Set camera's field of view to 80 ***
SetCameraFOV(1,80)

```

```

rem *** View scene ***
do
    HandleCamera()
    Sync()
loop

function HandleCamera()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            rem *** Turn camera ***
            case 37: //left cursor, turn camera left
                RotateCameraGlobalY(1,-1)
            endcase
            case 39: //right cursor, turn camera right
                RotateCameraGlobalY(1,1)
            endcase
        endselect
    endif
endfunction

```

Activity 23.32

In *FP3D*, the modified code for *HandleCamera()* is:

```

function HandleCamera()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            rem *** Turn camera ***
            case 37: //left cursor, turn camera left
                RotateCameraGlobalY(1,-1)
            endcase
            case 39: //right cursor, turn camera right
                RotateCameraGlobalY(1,1)
            endcase
            rem *** Move camera options ***
            case 87: //W key, Camera forward ***
                MoveCameraLocalZ(1,0.2)
            endcase
            case 83: //S key, Camera back ***
                MoveCameraLocalZ(1,-0.2)
            endcase
        endselect
    endif
endfunction

```

It is possible to move the camera through the walls.

Activity 23.33

In *FP3D*, the modified code for *HandleCamera()* is:

```

function HandleCamera()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            rem *** Turn camera ***
            case 37: //left cursor, turn camera left
                RotateCameraGlobalY(1,-1)
            endcase
            case 39: //right cursor, turn camera right
                RotateCameraGlobalY(1,1)
            endcase
            rem *** Move camera options ***
            case 87: //W key, Camera forward ***
                MoveCameraLocalZ(1,0.2)
            endcase
            case 83: //S key, Camera back ***
                MoveCameraLocalZ(1,-0.2)
            endcase
            case 38: //up cursor,tilt camera up
                if GetCameraAngleX(1) > -30
                    RotateCameraLocalX(1,-1)
                endif
            endcase
            case 40: //down cursor,tilt camera down
                if GetCameraAngleX(1) < 30
                    RotateCameraLocalX(1,1)
                endif
            endcase
        endselect
    endif
endfunction

```

```

endselect
endif
endfunction

```

With the camera pointing downward, moving forward causes the camera to move under the floor! With the camera tilted up, the camera moves up off the ground.

Activity 23.34

In *FP3D*, the modified code for *HandleCamera()* is:

```

function HandleCamera()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            rem *** Turn camera ***
            case 37: //left cursor, turn camera left
                RotateCameraGlobalY(1,-1)
            endcase
            case 39: //right cursor, turn camera right
                RotateCameraGlobalY(1,1)
            endcase
            rem *** Move camera options ***
            case 87: //W key, Camera forward ***
                rem *** Make camera parallel to floor ***
                angle = GetCameraAngleX(1)
                RotateCameraLocalX(1,-angle)
                rem *** Move camera ***
                MoveCameraLocalZ(1,0.2)
                rem *** Camera to original tilt ***
                RotateCameraLocalX(1,angle)
            endcase
            case 83: //S key, Camera back ***
                rem *** Make camera parallel to floor ***
                angle = GetCameraAngleX(1)
                RotateCameraLocalX(1,-angle)
                rem *** Move camera ***
                MoveCameraLocalZ(1,-0.2)
                rem *** Camera to original tilt ***
                RotateCameraLocalX(1,angle)
            endcase
            case 38: //up cursor,tilt camera up
                if GetCameraAngleX(1) > -30
                    RotateCameraLocalX(1,-1)
                endif
            endcase
            case 40: //down cursor,tilt camera down
                if GetCameraAngleX(1) < 30
                    RotateCameraLocalX(1,1)
                endif
            endcase
        endselect
    endif
endfunction

```

Activity 23.35

The main section of *FP3D* is now coded as:

```

rem *** First Person Perspective ***

rem *** Load textured walls ***
LoadObject(1,"Maze.obj",20)
LoadImage(1,"BricksLarge.png")
SetObjectImage(1,1,0)
rem *** Load floor ***
CreateObjectPlane(2,180,180)
RotateObjectLocalX(2,90)
SetObjectPosition(2,0,1,0)
SetObjectColor(2,220,180,180,0)

rem *** Create tree billboard ***
CreateObjectPlane(3,10,20)
LoadImage(3,"Tree.png")
SetObjectImage(3,3,0)
SetObjectTransparency(3,1)
SetObjectPosition(3,10,11,0)

rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,250,250,250)

rem *** Position camera ***
SetCameraPosition(1,5,5,-10)
SetCameraLookAt(1,5,5,-10,0)

```

```
rem *** Set camera's field of view to 80 ***
SetCameraFOV(1,80)
```

```
rem *** View scene ***
do
    HandleCamera()
    Sync()
loop
```

As you move round the tree, its two-dimensional nature becomes obvious.

Activity 23.36

In *FP3D*, the new function is coded as:

```
function HandleTree()
    rem *** Turn tree to look at camera ***
    SetObjectLookAt(3,GetCameraX(1),5,GetCameraZ(1),0)
endfunction
```

The final lines in function *HandleCamera()* become:

```
        endselect
        HandleTree()
    endif
endfunction
```

Although the tree always points towards the camera it appears to lift from the ground and tilt over.

Activity 23.37

In *FP3D*, the modified code for *HandleTree()* is:

```
function HandleTree()
    rem *** Turn tree to look at camera ***
    SetObjectLookAt(3,GetCameraX(1),5,GetCameraZ(1),0)
    rem *** Undo any rotation about the x-axis ***
    SetObjectRotation(3,0,GetObjectAngleY(3),0)
endfunction
```

Now the effect is complete, with the tree always facing the camera in an upright position.

Activity 23.38

No solution required.

Activity 23.39

In the main section of *FP3D*, the code required to create lights is changed to:

```
rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,10,10,10)
rem *** Add point light ***
CreateLightPoint(1,35,10,31,10,250,250,0)
```

This creates much duller lighting conditions throughout the model with a yellow light showing on the right of the maze.

Activity 23.40

In the main section of *FP3D*, change the line

```
CreateLightPoint(1,35,10,31,10,250,250,0)
```

to

```
CreateLightPoint(1,5,5,-10,10,250,250,0)
```

This is the initial position of the camera.

The final lines in function *HandleCamera()* become:

```
        LightPointPosition(1,GetCameraX(1),5,
        GetCameraZ(1))
        endselect
        HandleTree()
    endif
endfunction
```

Activity 23.41

The modified code for the main section of *FP3D* is:

```
rem *** First Person Perspective ***

rem *** Structure for point light data ***
type PointLightType
    colour    //250: yellow; 0: red
endtype
```

```
rem *** Global variables ***
rem *** Point light info ***
global light as PointLightType
light.colour = 250
```

```
rem *** Load textured walls ***
LoadObject(1,"Maze.obj",20)
LoadImage(1,"BricksLarge.png")
SetObjectImage(1,1,0)
rem *** Load floor ***
CreateObjectPlane(2,180,180)
RotateObjectLocalX(2,90)
SetObjectPosition(2,0,1,0)
SetObjectColor(2,220,180,180,0)
```

```
rem *** Create tree billboard ***
CreateObjectPlane(3,10,20)
LoadImage(3,"Tree.png")
SetObjectImage(3,3,0)
SetObjectTransparency(3,1)
SetObjectPosition(3,10,11,0)
```

```
rem *** Add directional light ***
CreateLightDirectional(1,0,-15,20,10,10,10)
rem *** Add point light ***
CreateLightPoint(1,5,5,-10,10,250,250,0)
```

```
rem *** Position camera ***
SetCameraPosition(1,5,5,-10)
SetCameraLookAt(1,5,5,-10,0)
rem *** Set camera's field of view to 80 ***
SetCameraFOV(1,80)
```

```
rem *** View scene ***
do
    HandleCamera()
    HandleLight()
    Sync()
loop
```

The code for *HandleLight()* is:

```
function HandleLight()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            rem *** Turn camera ***
            case 13: //Enter key, change colour
                if GetRawKeyPressed(key) = 1
                    light.colour = 250 - light.colour
                    SetLightPointColor(1,250,light.colour,0)
                endif
            endcase
        endselect
    endif
endfunction
```

Activity 23.42

The main section of *FP3D* now begins with the following code:

```
rem *** First Person Perspective ***

rem *** Structure for point light data ***
type PointLightType
    colour    //250: yellow; 0: red
    radius    //Radius of light
endtype

rem *** Global variables ***
rem *** Point light info ***
global light as PointLightType
light.colour = 1
light.radius = 10
```

The updated version of *HandleLight()* is:

```
function HandleLight()
    rem *** Get last key pressed ***
    key = GetRawLastKey()
    rem *** If key currently pressed, process it***
    if GetRawKeyState(key)=1
        select key
            rem *** Turn camera ***
            case 13: //Enter key, change colour
                if GetRawKeyPressed(key) = 1
                    light.colour = 250 - light.colour
                    SetLightPointColor(1,250,light.colour,0)
                endif
            endcase
            case 187: //+ key, increase light radius
                if light.radius < 5
                    inc light.radius
                endif
            endcase
            case 189: // - key. decrease light radius
                if light.radius > 5
                    dec light.radius
                endif
            endcase
        endselect
    endif
endfunction
```

Activity 23.43

In the main section of *FP3D*, create the floor using the following statements:

```
rem *** Load floor ***
CreateObjectPlane(2,180,180)
RotateObjectLocalX(2,90)
SetObjectPosition(2,0,1,0)
SetObjectColor(2,220,180,180,0)
rem *** Change floor's reflectivity ***
SetObjectLightMode(2,0)
```

Notice that the result of this is to have the floor reflect the original but unrealistic ambient light.

Activity 23.44

Since the ray cast is 15 units in length, a hit is detected well before the sphere reaches the plane.

To reduce the length of the cast to 1.5, change the line

```
hit = ObjectRayCast(1,-2,5,z#,-2,5,z#+15)
```

to

```
hit = ObjectRayCast(1,-2,5,z#,-2,5,z#+1.5)
```

Now the hit is detected just as the sphere reaches the plane.

Activity 23.45

No solution required.

Activity 23.46

The `do..loop` structure in *RayCast3D* should be changed to:

```
do
    rem *** Ray from centre of sphere 1.5 units ***
    hit = ObjectRayCast(1,x#,y#,z#, x#+xdist#,
        y#+ydist#, z#+zdist#)
    rem *** Display result ***
    SetTextString(1,Str(hit))
    rem *** Move sphere if no collision ***
    if hit = 0
        rem *** Move sphere along trajectory ***
        x# = x# + xstep#
        y# = y# + ystep#
        z# = z# + zstep#
        SetObjectPosition(2,x#,y#,z#)
    endif
    rem *** Update display ***
    Sync()
loop
```

Activity 23.47

RayCast3D should begin with the following lines:

```
rem *** 3D ray casting ***

rem *** Create plane ***
CreateObjectPlane(1,15,15)
rem *** Colour plane ***
SetObjectColor(1,200,100,100,255)
rem *** Move plane back ***
SetObjectPosition(1,0,0,15)

rem *** Make plane undetectable to collisions ***
SetObjectCollisionMode(1,0)
```

The sphere will move through the plane, since no collision is detected.

Activity 23.48

Within *RayCast3D*'s `do..loop` structure change the line

```
SetTextString(1,Str("Hit"))
```

to

```
SetTextString(1,"Hit at (" + Str(GetObjectRayCastX(0),2)
    + "," + Str(GetObjectRayCastY(0),2) + "," +
    + Str(GetObjectRayCastZ(0),2) + ")")
```

Activity 23.49

The modified code for *RayCast3D* is:

```
rem *** 3D ray casting ***

rem *** Load holed plane ***
LoadObject(1,"HoledPlane.obj",15)
rem *** Colour plane ***
SetObjectColor(1,200,100,100,255)
rem *** Move plane back ***
SetObjectPosition(1,0,0,15)

rem *** Create sphere ***
CreateObjectSphere(2,6,15,15)
rem *** Position sphere ***
SetObjectPosition(2,-2,5,-10)
rem *** Set up light ***
CreateLightDirectional(1,10,-10,20,200,200,200)

rem *** Position camera ***
SetCameraPosition(1,20,5,-10)
SetCameraLookAt(1,0,0,5,0)

rem *** Create text to display result ***
CreateText(1,"")

rem *** Sphere's initial position on all axes ***
x# = -2
y# = 5
z# = -10

rem *** Movement along each axis per frame ***
xstep# = 0.025
ystep# = -0.05
zstep# = 0.2

rem *** Calculate movement along each axis over a
distance of 3 units ***
multiplier# = 3 / (Sqrt(xstep#^2 + ystep#^2 +
    zstep#^2))
xdist# = xstep# * multiplier#
ydist# = ystep# * multiplier#
zdist# = zstep# * multiplier#

do
    rem *** Cast ray from centre of sphere 3 units ***
    hit = ObjectRayCast(1,x#,y#,z#, x#+xdist#,
        y#+ydist#,z#+zdist#)
    rem *** Display result ***
    SetTextString(1,"Hit at (" +
        + Str(GetObjectRayCastX(0),2) + "," +
        + Str(GetObjectRayCastY(0),2) + "," +
        + Str(GetObjectRayCastZ(0),2) + ")")
    rem *** Move sphere if no collision ***
    if hit = 0
        rem *** Move sphere along trajectory ***
```

```

x# = x# + xstep#
y# = y# + ystep#
z# = z# + zstep#

SetObjectPosition(2,x#,y#,z#)
endif
rem *** Update display ***
Sync()
loop

```

Even though its diameter is too large, the sphere still passes through the hole in the plane.

Activity 23.50

No solution required.

Activity 23.51

FP3D's *HandleCamera()* function should be recoded as:

```

function HandleCamera()
  rem *** Get last key pressed ***
  key = GetRawLastKey()
  rem *** If key currently pressed, process it***
  if GetRawKeyState(key)=1
    select key
      rem *** Turn camera ***
      case 37: //left cursor, turn camera left
        RotateCameraGlobalY(1,-1)
      endcase
      case 39: //right cursor, turn camera right
        RotateCameraGlobalY(1,1)
      endcase
      rem *** Move camera options ***
      case 87: //W key, Camera forward ***
        rem *** Make camera parallel to floor ***
        angle = GetCameraAngleX(1)
        RotateCameraLocalX(1,-angle)
        rem *** Move camera ***
        rem ** Get current position **
        oldx# = GetCameraX(1)
        oldy# = GetCameraY(1)
        oldz# = GetCameraZ(1)
        rem ** Move camera **
        MoveCameraLocalZ(1,0.2)
        rem ** Get new position **
        newx# = GetCameraX(1)
        newy# = GetCameraY(1)
        newz# = GetCameraZ(1)
        rem ** If collision, return camera to old
        position **
        hit = ObjectSphereCast(0,oldx#,oldy#,
        oldz#,newx#,newy#,newz#,1.5)
        if hit <> 0
          SetCameraPosition(1,oldx#,oldy#,oldz#)
        endif
        rem *** Return camera to original tilt ***
        RotateCameraLocalX(1,angle)
      endcase
      case 83: //S key, Camera back ***
        rem *** Make camera parallel to floor ***
        angle = GetCameraAngleX(1)
        RotateCameraLocalX(1,-angle)
        rem *** Move camera ***
        rem ** Get current position **
        oldx# = GetCameraX(1)
        oldy# = GetCameraY(1)
        oldz# = GetCameraZ(1)
        rem ** Move camera **
        MoveCameraLocalZ(1,-0.2)
        rem ** Get new position **
        newx# = GetCameraX(1)
        newy# = GetCameraY(1)
        newz# = GetCameraZ(1)
        rem ** If collision, return camera to old
        position **
        hit = ObjectSphereCast(0,oldx#,oldy#,
        oldz#,newx#,newy#,newz#,1.5)
        if hit <> 0
          SetCameraPosition(1,oldx#,oldy#,oldz#)
        endif
        rem *** Return camera to original tilt ***
        RotateCameraLocalX(1,angle)
      endcase
      case 38: //up cursor,tilt camera up
        if GetCameraAngleX(1) > -30

```

```

        RotateCameraLocalX(1,-1)
      endif
    endcase
  case 40: //down cursor,tilt camera down
    if GetCameraAngleX(1) < 30
      RotateCameraLocalX(1,1)
    endif
  endcase
endselect
HandleTree()
rem *** Reposition light ***
SetLightPointPosition(1, GetCameraX(1),5,
GetCameraZ(1))
endif
endfunction

```

Activity 23.52

There are three hits before the sphere stops moving.

Activity 23.53

No solution required.

Activity 23.54

Modified code for *SphereSlide3D* :

```

rem *** Demonstrating Bounce Marks ***

rem *** Create first wall ***
CreateObjectPlane(1,20,10)
rem *** Create second wall at right angles to first
***
CreateObjectPlane(2,20,10)
RotateObjectLocalY(2,90)
SetObjectPosition(2,10,0,-10)
rem *** Create floor ***
CreateObjectPlane(3,20,20)
RotateObjectLocalX(3,90)
SetObjectPosition(3,0,-5,-10)

rem *** Create directional light ***
CreateLightDirectional(1,5,-10,0,255,255,255)

rem *** Position Camera ***
SetCameraPosition(1,-10,5,-50)
SetCameraLookAt(1,10,0,0,0)

rem *** Create sphere ***
CreateObjectSphere(4,3,15,15)

rem *** sphere's position ***
x# = -5
y# = 4
z# = -15
SetObjectPosition(4,x#,y#,z#)

rem *** Sphere's movement vector offsets ***
xoff# = 0.075
yoff# = -0.05
zoff# = 0.15

rem *** Switch off the sphere's collision detection
***
SetObjectCollisionMode(4,0)

rem *** Create Bounce Mark plane ***
LoadImage(1,"RedCircle.png")
CreateObjectPlane(5,1.5,1.5)
SetObjectImage(5,1,0)
SetObjectTransparency(5,1)
SetObjectCollisionMode(5,0)
SetObjectPosition(5,0,1200,0)

rem *** Object number ***
objno = 5
do
  rem *** Get current position of sphere ***
  oldx# = GetObjectX(4)
  oldy# = GetObjectY(4)
  oldz# = GetObjectZ(4)

  rem *** Move the sphere object ***
  MoveObjectLocalX(4,xoff#)
  MoveObjectLocalY(4,yoff#)
  MoveObjectLocalZ(4,zoff#)

```



```

rem *** Get sphere's new position ***
x# = GetObjectX(4)
y# = GetObjectY(4)
z# = GetObjectZ(4)
rem *** Perform sphere cast between old and new
    ↪position ***
hit = ObjectSphereCast(0,oldx#,oldy#,oldz#
    ↪,x#,y#,z#,1.5)
rem *** If hit...
if hit > 0
    rem *** Add a bounce mark ***
    inc objno
    InstanceObject(objno,5)
    rem *** Get the collision point ***
    hitx# = GetObjectRayCastX(0)
    hity# = GetObjectRayCastY(0)
    hitz# = GetObjectRayCastZ(0)
    rem *** Position bounce mark at collision
    ↪point ***
    SetObjectPosition(objno,hitx#,hity#,hitz#)
    rem *** Orientate bounce mark along the
    ↪normal ***
    ox# = hitx# + GetObjectRayCastNormalX(0)
    oy# = hity# + GetObjectRayCastNormalY(0)
    oz# = hitz# + GetObjectRayCastNormalZ(0)
    SetObjectLookat(objno,ox#,oy#,oz#,0)
    rem *** Reposition the sphere at the point of
    ↪collision ***
    SetObjectPosition(4,hitx#,hity#,hitz#)
    rem *** Change sphere's trajectory ***
    xoff# = GetObjectRayCastBounceX(0)
    yoff# = GetObjectRayCastBounceY(0)
    zoff# = GetObjectRayCastBounceZ(0)
endif
Sync()
loop

```

Activity 23.55

No solution required.

Activity 23.56

The main section of *3DToScreen* should be changed to:

```

rem *** 3D Coords to Screen Coords ***

rem *** Create sphere ***
CreateObjectSphere(1,0.5,15,15)

rem *** Create directional light ***
CreateLightDirectional(1,5,-10,0,255,255,255)

rem *** Position Camera ***
SetCameraPosition(1,0,0,-50)
SetCameraLookat(1,0,0,-50,0)

rem *** Create text to display results ***
CreateText(1,"")
SetTextSize(1,2)

rem *** Load label image ***
LoadImage(1,"BubbleSphere.png")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpriteSize(1,10,-1)
rem *** Sprite offset required so pointer part ***
rem *** of the image is over sphere ***
yoffset = -9

rem *** Allow camera movement ***
do
    HandleCamera()
    SetTextString(1,"The centre of the sphere maps
    ↪to screen coords \"
    ↪+Str(GetScreenXFrom3D(0,0,0),2)+\", \"
    ↪+Str(GetScreenYFrom3D(0,0,0),2)+\"")
    rem *** Position sprite ***
    SetSpritePosition(1,GetScreenXFrom3D(0,0,0),
    ↪GetScreenYFrom3D(0,0,0)+yoffset)
    Sync()
loop

```

Activity 23.57

No solution required.

Activity 23.58

No solution required.

Activity 23.59

Using the shader gives a brighter image - similar to that obtained from the ambient light.

Memory Blocks

In this Chapter:

- ☐ What is a Memory Block?
- ☐ Accessing a Memory Block
- ☐ Using Memory Blocks to Create Data Structures
- ☐ Memory Blocks and Files
- ☐ Image Memory Blocks
- ☐ Manipulating an Image in a Memory Block
- ☐ Creating a New Image within a Memory Block
- ☐ The Mandelbrot Set Project

Accessing Memory

Introduction

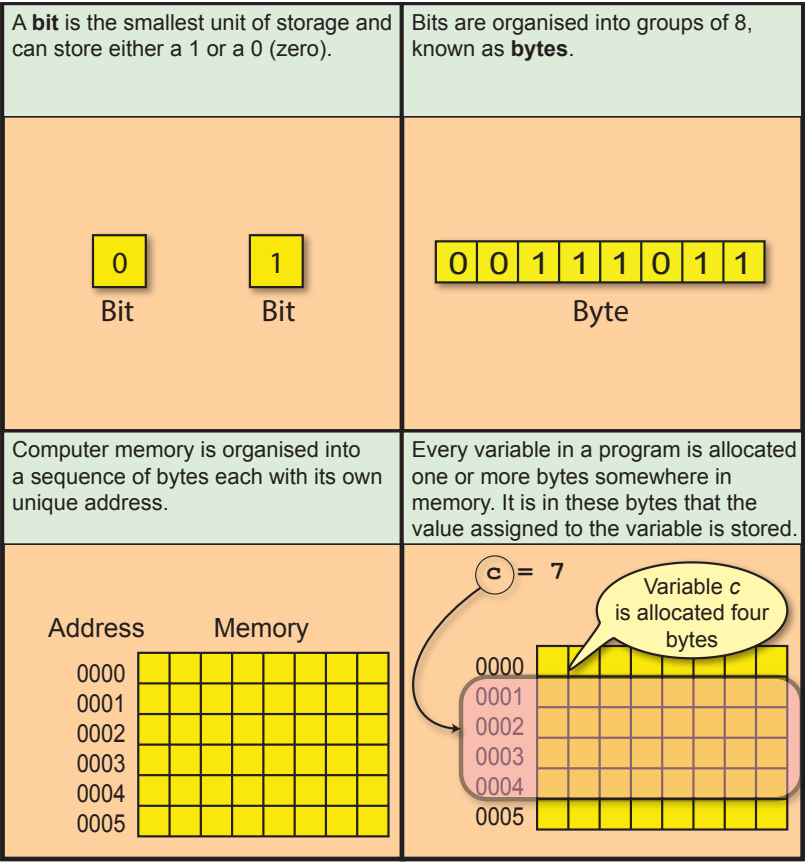
The basic storage unit in computer memory is the **bit**. A bit can store a single binary digit (0 or 1). In a computer’s main memory, these bits are organised into groups of 8 known as **bytes**. Every byte is allocated a unique memory address. These addresses are allocated in sequential order. Hence the first location is given the address 0, the next address 1, etc.

When we create a variable with a statement such as

```
lives = 3
```

the computer reserves a few bytes of memory for that variable (see FIG-24.1) and it’s in those bytes that the variable’s contents are stored.

FIG-24.1
Memory Organisation
and Allocation



Most of the time, finding out which locations have been allocated to a variable is of no interest to us since we can access its contents by using the variable’s name.

It’s not only variables that are assigned space in memory, so are images, sounds, 3D objects and other elements within a program.

However, it is also possible to reserve memory space explicitly within your code using the memory block (shortened to **memblock**) commands.

Memory Block Statements

CreateMemblock()

To reserve a specified number of bytes within a device’s memory, use the `CreateMemblock()` statement (see FIG-24.2).

FIG-24.2

CreateMemblock()

Format 1

CreateMemblock (id , ibytes)

Format 2

integer CreateMemblock (ibytes)

where

- id** is an integer value giving the ID to be assigned to the memory block being reserved.
- ibytes** is an integer value (1 to 100,000,000) giving the number of bytes to be reserved.

Using the first format requires you to specify the ID to be assigned; using the second format creates an ID automatically and returns that value.

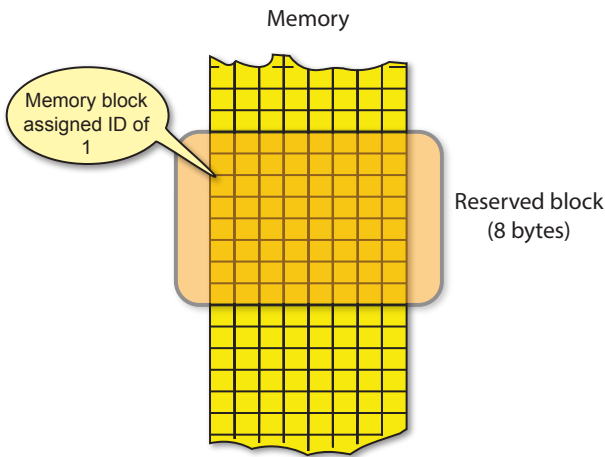
If we were to execute the statement

`CreateMemblock(1,8)`

then a block of 8 bytes, assigned the ID value 1, would be reserved in the device’s memory (see FIG-24.3).

FIG-24.3

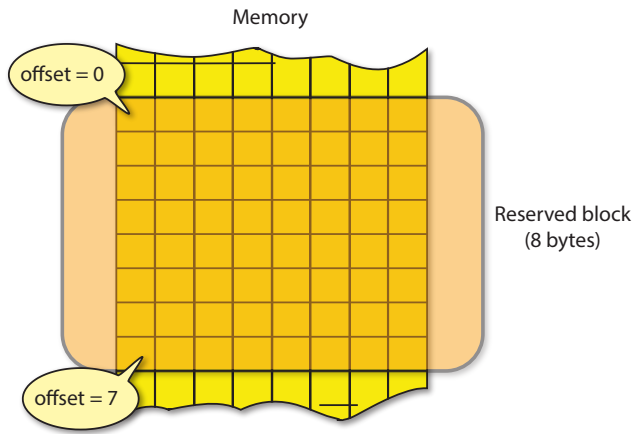
Creating a Memory Block



There is no need to know the exact address of this reserved memory, but we do need some method of specifying which location within the reserved area we want to access. This is done by specifying an offset from the start of the block. For example, the first byte has an offset of zero, the last an offset of 7 (see FIG-24.4).

FIG-24.4

Addressing within a
Memory Block



SetMemblockByte()

Once we have reserved our memory block, we can write a value to a specified byte within that block using the `SetMemblockByte()` statement (see FIG-24.5).

FIG-24.5

SetMemblockByte()

`SetMemblockByte ((id , ioff , iv))`

where

- id** is an integer value giving the ID previously assigned to a reserved memory block.
- ioff** is an integer value giving the offset within the memblock to which the value is to be written.
- iv** is an integer (0 to 255) giving the value to be stored in the memory block.

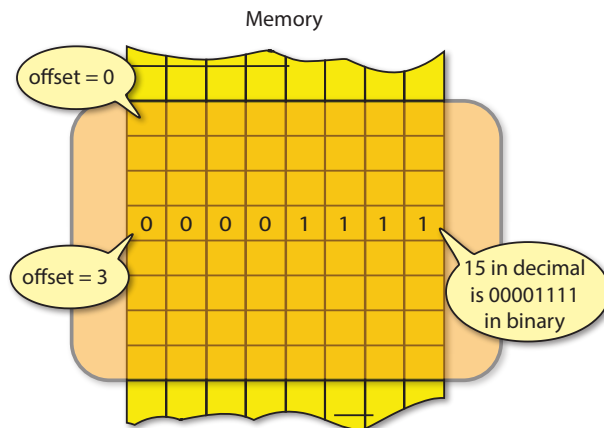
For example, executing the statements

```
CreateMemblock (1,8)  
SetMemblockByte (1,3,15)
```

would have the effect shown in FIG-24.6.

FIG-24.6

Writing a Byte to a
Memory Block



SetMemblockShort()

A two-byte integer value can be written to a memory block using the `SetMemblockShort()` statement (see FIG-24.7).

FIG-24.7

SetMemblockShort()

`SetMemblockShort ((id , ioff , iv))`

where

- id** is an integer value giving the ID previously assigned to a reserved memory block.
- ioff** is an integer value giving the offset within the memblock to which the value is to be written. Must be a multiple of 2. (0, 2, 4, etc.)
- iv** is an integer (-32,768 to 32,767) giving the value to be stored in the memory block.

Notice that the offset value (*ioff*) must be a multiple of 2; you cannot start a short integer value at an odd offset location.

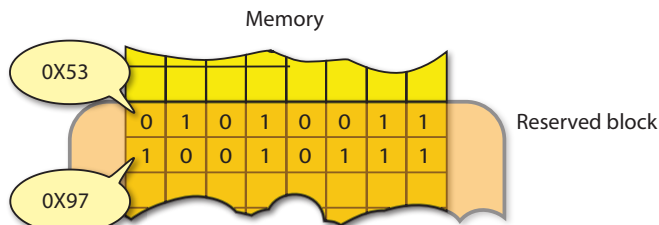
Another peculiarity of the storage method is that the bytes of the value specified are stored in reverse order. For example, if we execute the statement

```
SetMemblockShort(1,0,0X5397) //Value given in hexadecimal
```

we might expect the value to be stored as shown in FIG-24.8

FIG-24.8

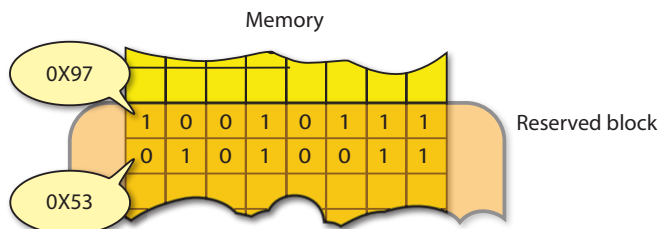
Expected Storage Format



but, in fact the bytes are stored in the reverse order (see FIG-24.9).

FIG-24.9

Actual Storage Format



Generally, this reversal of the bytes that make up the value will be transparent to you since the processor itself handles both the reversal when storing the value and restoration of the correct byte order when the value is later retrieved. The only time you are likely to be aware of this strange setup is if you were to write a `short` value and then read the same value back a byte at a time.

This storage format is known as **little endian** since the least-significant byte is stored first.

SetMemblockInt()

To write a four-byte integer value to a memory block, use `SetMemblockInt()` (see FIG-24.10).

FIG-24.10

SetMemblockInt()

`SetMemblockInt` ((`id` , `ioff` , `iv`)

where

id	is an integer value giving the ID previously assigned to a reserved memory block.
ioff	is an integer value giving the offset within the memblock to which the value is to be written. Must be a multiple of 4. (0, 4, 8, etc.)
iv	is an integer (-2,147,483,648 to 2,147,483,647) giving the value to be stored in the memory block.

Again, the bytes of the value are stored in reverse order, so executing

```
SetMemblockInt(1,0,0xF612ACD3)
```

would store the bytes in the order

D3
AC
12
F6

SetMemblockFloat()

A four-byte real value can be stored in a memory block using the `SetMemblockFloat()` statement (see FIG-24.11).

FIG-24.11

SetMemblockFloat()

`SetMemblockFloat` ((`id` , `ioff` , `v`)

where

id	is an integer value giving the ID previously assigned to a reserved memory block.
ioff	is an integer value giving the offset within the memblock to which the value is to be written. Must be a multiple of 4. (0, 4, 8, etc.)
v	is a real value giving the number to be stored in the memory block.

Real values are stored using a sign bit, bias exponent and significant.

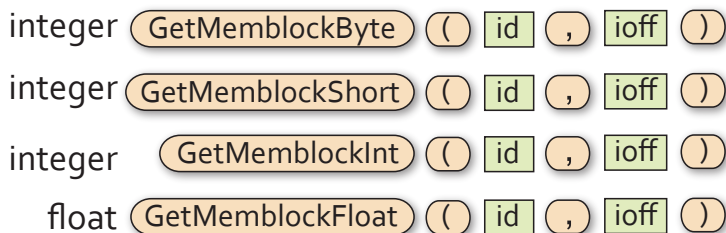
GetMemblockByte(), GetMemblockShort(), GetMemblockInt() and GetMemblockFloat()

AGK offers a companion set of functions for reading byte, short, integer and real

values from a memblock. These are `GetMemblockByte()`, `GetMemblockShort()`, `GetMemblockInt()` and `GetMemblockFloat()` (see FIG-24.12).

FIG-24.12

`GetMemblockByte()`
`GetMemblockShort()`
`GetMemblockInt()`
`GetMemblockFloat()`



where

id is an integer value giving the memblock ID.

ioff is an integer value giving the offset within the memblock from which a value is to be read. For short values this must be a multiple of 2; for integer and real values, a multiple of 4.

The program in FIG-24.13 demonstrates the use of a memblock to store byte, short, integer and real values as well as displaying those stored values on the screen.

FIG-24.13

Writing and Reading a
Memory Block

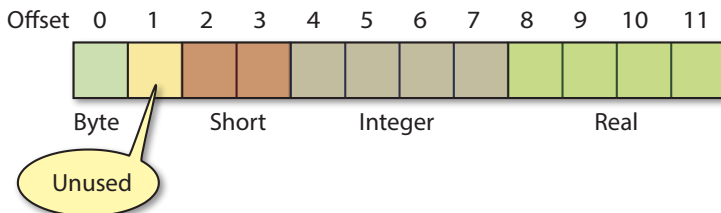
```
rem *** Data Memblock ***

rem *** Set up memblock ***
CreateMemblock(1,12)
rem *** Store byte value in first byte ***
SetMemblockByte(1,0,255)
rem *** Store short value in third and fourth bytes ***
SetMemblockShort(1,2,1200)
rem *** Store integer in 5th to 8th bytes ***
SetMemblockInt(1,4,7654321)
rem *** Store real in 9th to 12th bytes ***
SetMemblockFloat(1,8,3.14159)
rem *** display each value ***
do
  Print(GetMemblockByte(1,0))
  Print(GetMemblockShort(1,2))
  Print(GetMemblockInt(1,4))
  Print(GetMemblockFloat(1,8))
  Sync()
loop
```

The program's memblock layout is shown in FIG-24.14 (each cell represents 1 byte).

FIG-24.14

Memory Block Data
Layout



Activity 24.1

Why is the second byte of the memblock as shown in FIG-24.14 unused?

Activity 24.2

Start a new project called *MemblockData* and implement the code given in FIG-24.13.

Test and save your program.

GetMemblockExists()

You can check that a memblock with a specified ID currently exists using `GetMemblockExists()` (see FIG-24.15).

FIG-24.15

GetMemblockExists()

integer `GetMemblockExists` ((id))

where

id is an integer value giving the ID to be checked.

The function returns 1 if a memblock of the specified ID exists, otherwise zero is returned.

DeleteMemblock()

To free up the memory reserved by an existing memblock, use `DeleteMemblock()` (see FIG-24.16).

FIG-24.16

DeleteMemblock()

`DeleteMemblock` ((id))

where

id is an integer value giving the ID of the memblock to be deleted.

GetMemblockSize()

To discover the size of an existing memblock, use `GetMemblockSize()` (see FIG-24.17).

FIG-24.17

GetMemblockSize()

integer `GetMemblockSize` ((id))

where

id is an integer value giving the ID of the memblock whose size is to be determined.

The function returns the number of bytes that have been allocated to the specified memblock.

Storing Characters and Strings in a Memory Block

Although there are commands to store integer and real values in a memblock, there are none for storing characters or strings.

Storing a Character

Storing a single character is relatively simple. All we have to do is use the `SetMemblockByte()` statement, having converted the character to its numeric ASCII value using the `Asc()` function. For example we could store the letter A in the first byte of an existing memblock (whose ID is 1) using the line:

```
SetMemblockByte(1,0,Asc("A"))
```

When reading the letter back from the memblock, we need to convert the number back to a character using `Chr()` :

```
letter$ = Chr(GetMemblockByte(1,0))
```

Storing a String

In order to store a string in a memblock, we must first know the number of characters in that string. This will determine the size of the memblock which needs to be created.

If the string we wish to store is in the variable *name\$*, then we would create the memblock using the line

```
id = CreateMemblock(Len(name$))
```

To store the string in the memblock, we must write each character of the string in turn, remembering to first convert it to a numeric value. We can do this with the following code:

```
for c = 1 to Len(name$)
  SetMemblockByte(id,c-1,Asc(Mid(name$,c,1)))
next c
```

Reading the string from the memblock requires the characters to be retrieved a byte at a time, converting each back from a number to a character. This is achieved by the following code:

```
result$ = ""
for c = 0 to GetMemblockSize(id)-1
  result$ = result$ + Chr(GetMemblockByte(id,c))
next c
```

The program in FIG-24.18 demonstrates the storage and retrieval of a single string from a memblock. The code contains three functions:

`CreateMemString()` creates a memblock containing a specified string and returns the ID assigned to the memblock.

`SetMemString()` stores a specified string in the memblock.

`GetMemString()` returns the string held in a memblock.

FIG-24.18

Storing a String in a
Memory Block

```

rem *** Using a memblock to store a string ***

rem *** Set up string and display its contents ***
mystring = CreateMemString("Klaatu barada nikto")
do
    Print(GetMemString(mystring))
    Sync()
loop

function CreateMemString(s$)
    rem *** Create memblock for string ***
    id = CreateMemblock(Len(s$))
    rem *** Copy chars from string to memblock ***
    for c = 1 to Len(s$)
        SetMemblockByte(id,c-1,Asc(Mid(s$,c,1)))
    next c
endfunction id

function SetMemString(id,s$)
    rem *** If block exists, delete it ***
    if GetMemblockExists(id) = 1
        DeleteMemblock(id)
    endif
    rem *** Create memblock for string ***
    CreateMemblock(id,Len(s$))
    rem *** Copy chars from string to memblock ***
    for c = 1 to Len(s$)
        SetMemblockByte(id,c-1,Asc(Mid(s$,c,1)))
    next c
endfunction

function GetMemString(id)
    rem *** Return empty string if memblock does not exist ***
    if GetMemblockExists(id) = 0
        exitfunction ""
    endif
    rem *** Start with empty string ***
    result$ = ""
    for c = 0 to GetmemBlockSize(id)-1
        result$ = result$ + Chr(GetMemblockByte(id,c))
    next c
endfunction result$

```

Notice that the function `SetMemString()` deletes any existing memblock allocation. This is necessary because the new string will almost certainly require a different number of bytes in the memblock than the previous string held there.

Activity 24.3

Start a new project called *MemblockString* and implement the code given in FIG-24.18.

Test and save your program.

The other point of interest in the code is that each function accesses the memblock by making the ID of the memblock a parameter. This approach is identical to that used by the various AGK statements used to handle resources such as images, sprites and 3D objects.

Using a Memory Block as an Array

To emulate an array within a memblock, we again need functions to create the memblock, assign a value to a particular “cell”, and retrieve the value from a given “cell”.

The code in FIG-24.19 contains functions to set up and access an integer array. The functions are then used to create a 10-element memblock array and assign the cells of the array the values 3 to 30 (in sequence). The contents of the cells are then displayed on the screen.

FIG-24.19

Emulating an Array
Using a Memblock

```
rem *** Integer array as memblock ***

rem *** Create a ten element memblock array ***
mydata = CreateMemIntArray(10)
rem *** Set values in array to 3 - 30 ***
for c = 0 to 9
    SetMemIntArrayCell(mydata,c,(c+1)*3)
next c

do
    rem *** Display the contents of array ***
    for c = 0 to 9
        Print (GetMemIntArrayCell(mydata,c))
    next c
    Sync()
loop

function CreateMemIntArray(size)
    rem *** Create memblock of required size ***
    id = CreateMemblock(size*4)
    rem *** Set all cells to zero ***
    for c = 0 to size-1
        SetMemblockInt(id,c*4,0)
    next c
endfunction id

function SetMemIntArrayCell(id,idx,v)
    rem *** Exit if memblock does not exist ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Exit if array index invalid ***
    if idx < 0 or idx > GetMemblockSize(id)/4
        exitfunction
    endif
    SetMemblockInt(id,idx*4,v)
endfunction

function GetMemIntArrayCell(id,idx)
    rem *** Exit if memblock does not exist ***
    if GetMemblockExists(id)=0
        exitfunction 0
    endif
    rem *** Exit if array index invalid ***
    if idx < 0 or idx > GetMemblockSize(id)/4
        exitfunction 0
    endif
    result = GetMemblockInt(id,idx*4)
endfunction result
```

Activity 24.4

Start a new project called *MemblockArray* and implement the code given in FIG-24.19.

Test and save your program.

Using a Memory Block as a Record Structure

Simple Records

In standard AGK we create a record by first defining a type

```
type ColourType
  red
  green
  blue
endtype
```

and then creating a variable of that type:

```
tint as ColourType
```

If we use a memblock to serve the same purpose, we need to start by calculating the number of bytes required:

Field	Type	Size
red	= byte	= 1 byte
green	= byte	= 1 byte
blue	= byte	= 1 byte
Total		3 bytes

Next we need to remember the offset within the memblock where each field starts:

```
red    = 0 offset
green  = 1 offset
blue   = 2 offset
```

Now we are ready to replace our record structure with a memblock and a set of corresponding functions allowing us to set and get each of the record's fields. The code for this is shown in FIG-24.20.

FIG-24.20

Storing a Record in a
Memory Block

```
rem *** Creating a Record-Type Structure using Memblocks ***

remstart
Below is the conceptual structure being created in the memblock
type ColourType
  red  byte
  green byte
  blue byte
endtype
remend
```



FIG-24.20

(continued)

Storing a Record in a
Memory Block

```
rem *** Create a memblock of this type ***
mycolour = CreateMemColourType(255,200,100)
rem *** Display the contents of the memblock
do
    Print("Red    : "+Str(GetMemColourRed(mycolour)))
    Print("Green  : "+Str(GetMemColourGreen(mycolour)))
    Print("Blue   : "+Str(GetMemColourBlue(mycolour)))
    Sync()
loop

rem *** Creates and intialises the structure ***
rem *** returns the ID ***
function CreateMemColourType(r,g,b)
    rem *** Create memblock for record ***
    id = CreateMemblock(3)
    rem *** Store initial colours staying in range 0 to 255 ***
    SetMemblockByte(id,0,r mod 256)
    SetMemblockByte(id,1,g mod 256)
    SetMemblockByte(id,2,b mod 256)
endfunction id

rem *** Change all three colour elements ***
function SetMemColour(id,r,g,b)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Set red component ***
    SetMemblockByte(id,0,r mod 256)
    rem *** Set green component ***
    SetMemblockByte(id,1,g mod 256)
    rem *** Set blue component ***
    SetMemblockByte(id,2,b mod 256)
endfunction

rem *** Change red only ***
function SetMemColourRed(id,r)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Set red component ***
    SetMemblockByte(id,0,r mod 256)
endfunction

rem *** Change green only ***
function SetMemColourGreen(id,g)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Set green component ***
    SetMemblockByte(id,1,g mod 256)
endfunction
```



FIG-24.20

(continued)

Storing a Record in a
Memory Block

```

rem *** Change blue only ***
function SetMemColourBlue(id,r)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Set blue component ***
    SetMemblockByte(id,2,b mod 256)
endfunction

rem *** Get current red setting ***
function GetMemColourRed(id)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction 0
    endif
    rem *** Retrieve red value ***
    result = GetMemblockByte(id,0)
endfunction result

rem *** Get current green setting ***
function GetMemColourGreen(id)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction 0
    endif
    rem *** Retrieve red value ***
    result = GetMemblockByte(id,1)
endfunction result

rem *** Get current blue setting ***
function GetMemColourBlue(id)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction 0
    endif
    rem *** Retrieve red value ***
    result = GetMemblockByte(id,2)
endfunction result

```

Activity 24.5

Start a new project called *MemblockColour* and implement the code in FIG-24.20.

Modify the code so that the initial colour values are changed before being displayed.

Test and save your project.

By using the memblock approach, we can update fields within a record - something that AGK does not guarantee when performed within a function.

Activity 24.6

Start a new project called *MemblockPoint* and create a memblock which is equivalent to the definition

```
type PointType
  x as float
  y as float
endtype
```

The code should contain the following functions:

<code>CreateMemPointType(x#,y#)</code>	creates memblock and sets values in the <i>x</i> and <i>y</i> fields. Returns memblock ID.
<code>SetMemPoint(id,x#,y#)</code>	sets values in <i>x</i> and <i>y</i> fields. No return value.
<code>GetMemPointX(id)</code>	returns value of <i>x</i> field.
<code>GetMemPointY(id)</code>	returns value of <i>y</i> field.

Test and save your program.

Arrays within Records

A `type` definition that is not allowed in AGK BASIC is one which contains an array. For example, let's say a character in a role-playing game is allowed to carry up to 10 items but the weight of each item affects the character's speed. The structure we would like to create might be defined as:

```
type RPGCharacterType
  charactersnum as integer
  dim weights[9] as float
endtype
```

To overcome this AGK BASIC restriction, we can make use of a memblock. Running through the calculations we get:

Bytes required

charactersnum	4
weights[0]-[9]	40
Total	44

Offsets

charactersnum	0
weights[0]	4
weights[1]	8
weights[2]	12
...	...
weights[9]	40

Although the array subscripts run from 0 to 9, most people are happier using the values 1 to 10 and it is this range that we will use when implementing the routines to handle the memblock.

This time we will need only five routines. These are:

<code>CreateMemRPGCharacterType(chnum)</code>	creates <i>RPGCharacterType</i> memblock
---	--

```

SetMemRPGCharNum(id, chnum)
SetMemRPGWeight(id, idx, w#)
GetMemRPGCharId(id)
GetMemRPGWeight(id,idx)

```

and sets *charactersnum* to *chnum*. All weights are set to zero. Returns memblock ID.

sets value of *charactersnum* to *chnum*.

sets value of *weights[idx]* to *w#*.

returns value of *charactersnum*.

returns value of *weights[idx]*.

The program in FIG-24.21 demonstrates each of these functions.

FIG-24.21

Using a Memory Block
to Emulate a Record
Containing an Array

```

rem *** Arrays within a Memblock ***

remstart *** Conceptual structure ***
type RPGCharacterType
  charcaternum as integer
  dim weights[9] as float
endtype
remend

rem *** Create a RPGCharacterType object ***
mychar = CreateMemRPGCharacterType(1234)

rem *** Set the first weight ***
SetMemRPGWeight(mychar,1,98.7)

rem *** Display the contents of the RPGCharacter object ***
do
  Print("Character's number : "+Str(GetMemRPGCharNum(mychar)))
  for c = 1 to 10
    Print("Weight "+Str(c)+"      : "+
      ↳Str(GetMemRPGWeight(mychar,c), 2))
  next c
  Sync()
loop

rem *** Create RPG memblock and set charcater's ***
rem *** number to chnum and all weights to zero ***
function CreateMemRPGCharacterType(chnum)
  rem *** Create memblock for record ***
  id = CreateMemblock(44)
  rem *** Store character's number ***
  SetMemblockInt(id,0,chnum)
  rem *** Set all weights to zero ***
  for c = 1 to 10
    SetMemblockFloat(id,c*4,0)
  next c
endfunction id

rem *** Set character's number ***
function SetMemRPGCharNum(id,chnum)
  rem *** if memblock does not exist, exit ***
  if GetMemblockExists(id)=0
    exitfunction
  endif
  rem *** Set char num component ***
  SetMemblockInt(id,0,chnum)
endfunction

```



FIG-24.21

(continued)

Using a Memory Block
to Emulate a Record
Containing an Array

```

rem *** Set weight in weight[idx] ***
function SetMemRPGWeight(id,idx,w#)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Set weight[idx] component ***
    SetMemblockFloat(id,idx*4,w#)
endfunction

rem *** Get character's number ****
function GetMemRPGCharNum(id)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction 0
    endif
    rem *** Retrieve character's number value ***
    result = GetMemblockInt(id,0)
endfunction result

rem *** Get weight in weight[idx] ***
function GetMemRPGWeight(id,idx)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction 0.0
    endif
    rem *** Retrieve weight[idx] ***
    result# = GetMemblockFloat(id,idx*4)
endfunction result#

```

Activity 24.7

Start a new project called *MemblockRecArray* and implement the code given in FIG-24.21.

Try saving values in other elements of the *weights* array.

Test and save your project.

What other precondition should be added to the `SetMemRPGWeight()` function?

Nested Records

Assuming we have already defined a structure for types `PointType` and `ColourType`, then we could define the data structure for a line as:

```

type LineType
    start as PointType    //Start point of line
    finish as PointType   //End point of line
    colour as ColourType //Line colour
endtype

```

And although we could create such a structure without resorting to the use of memblocks, the limitation hanging over such an approach is that it would not be possible to update a record of this structure within a function.

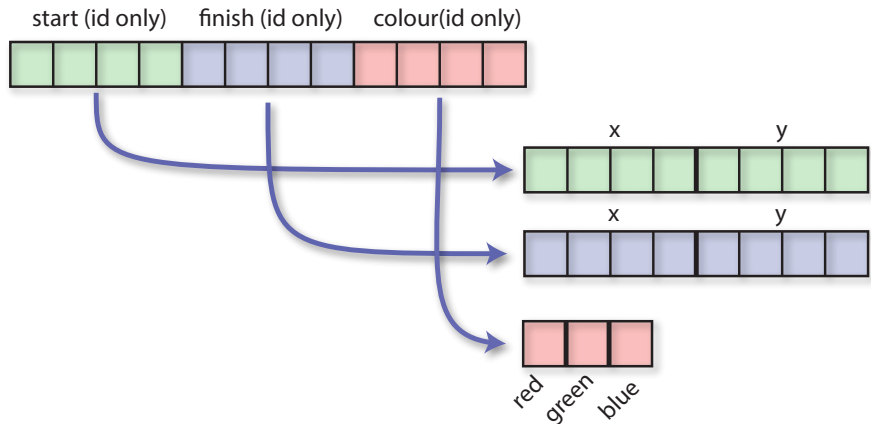
On the other hand, if we have previously defined **PointType** and **ColourType** using memblocks, we can make use of these to create a **LineType** which is also based on a memblock.

However, when we create a record structure, **LineType**, containing fields which are themselves record structures *start*, *finish* and *colour*, then these fields are set up as simple integers which will hold the IDs of the actual data structures (see FIG-24.22).

FIG-24.22

How Nested Records are Implemented

LineType (Internal Structure)



With this structure, creating a **LineType** structure also requires us to create the data areas where the coordinates and colour information will be stored. Hence, the function **CreateMemLineType()** would be coded as:

```
function CreateMemLineType()
    rem *** Create main memblock ***
    id = CreateMemblock(12)
    rem *** Create memblock for fields in record ***
    startid = CreateMemPointType(0,0)
    finishid = CreateMemPointType(0,0)
    colourid = CreateMemColourType(255,255,255)
    rem *** Store IDs in main memblock ***
    SetmemblockInt(id,0,startid)
    SetMemblockInt(id,4,finishid)
    SetMemblockInt(id,8,colourid)
endfunction id
```

Any function assigning a value to a field within a **LineType** structure would make use of the appropriate existing function already defined within the **PointType** or **ColourType** structures. Hence, the function **SetMemLineStart()** would be defined as:

```
function SetMemLineStart(id, x#, y#)
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    startid = GetMemblock(id,0)
    SetMemPoint(startid,x#,y#)
endfunction
```

A complete program which demonstrates the use of a **LineType** memblock structure is shown in FIG-24.23.

FIG-24.23

Using a Memory Block
to Emulate a Nested
Records Structure

```

rem *** Using Memblocks to Create a Nested Record ***

rem *** Create a LineType structure ***
myline = CreateMemLineType()

rem *** Assign start, finish and colour to line ***
SetMemLineStart(myline,20.5,34.8)
SetMemLineFinish(myline,76.2,51.6)
SetMemLineColour(myline,255,100,200)

rem *** Display the details of the line ***
do
    Print("Line start  : (" + Str(GetMemLineStartX(myline),2) +
    ↵ ", " + Str(GetMemLineStartY(myline),2) + ")")
    Print("Line finish  : (" + Str(GetMemLineFinishX(myline),2) +
    ↵ ", " + Str(GetMemLineFinishY(myline),2) + ")")
    Print("Line colour  : " + Str(GetMemLineColourRed(myline)) +
    ↵ ", " + Str(GetMemLineColourGreen(myline)) +
    ↵ ", " + Str(GetMemLineColourBlue(myline)))
    Sync()
loop

rem *** Creates a linetype with default values ***
rem *** Start=(0,0); finish = (0,0) ***
rem *** colour = 255,255,255 (white) ***
function CreateMemLineType()
    rem *** Create main memblock ***
    id = CreateMemblock(12)
    rem *** Create memblock for fields in record ***
    startid = CreateMemPointType(0,0)
    finishid = CreateMemPointType(0,0)
    colourid = CreateMemColourType(255,255,255)
    rem *** Store IDs in main memblock ***
    SetMemblockInt(id,0,startid)
    SetMemblockInt(id,4,finishid)
    SetMemblockInt(id,8,colourid)
endfunction id

rem *** Sets start point to (x#,y#) ***
function SetMemLineStart(id,x#,y#)
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    startid = GetMemblockInt(id,0)
    SetMemPoint(startid,x#,y#)
endfunction

rem *** Sets finish point to (x#,y#) ***
function SetMemLineFinish(id,x#,y#)
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    finishid = GetMemblockInt(id,4)
    SetMemPoint(finishid,x#,y#)
endfunction

```



FIG-24.23

(continued)

Using a Memory Block
to Emulate a Nested
Records Structure

```

rem *** Sets the line's colour to r,g,b ***
function SetMemLineColour(id,r,g,b)
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    colourid = GetMemblockInt(id,8)
    SetMemColour(colourid,r,g,b)
endfunction

rem *** Returns the X coord of the start point ***
function GetMemLineStartX(id)
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    startid = GetMemblockInt(id,0)
    result# = GetMemPointX(startid)
endfunction result#

rem *** Returns the Y coord of the start point ***
function GetMemLineStartY(id)
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    startid = GetMemblockInt(id,0)
    result# = GetMemPointY(startid)
endfunction result#

rem *** Returns the X coord of the finish point ***
function GetMemLineFinishX(id)
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    finishid = GetMemblockInt(id,4)
    result# = GetMemPointX(finishid)
endfunction result#

rem *** Returns the Y coord of the finish point ***
function GetMemLineFinishY(id)
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    finishid = GetMemblockInt(id,4)
    result# = GetMemPointY(finishid)
endfunction result#

rem *** Returns the red component of the line's colour ***
function GetMemLineColourRed(id)
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    colourid = GetMemblockInt(id,8)
    result = GetMemColourRed(colourid)
endfunction result

```



FIG-24.23

(continued)

Using a Memory Block
to Emulate a Nested
Records Structure

```

rem *** Returns the green component of the line's colour ***
function GetMemLineColourGreen(id)
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    colourid = GetMemblockInt(id,8)
    result = GetMemColourGreen(colourid)
endfunction result

rem *** Returns the blue component of the line's colour ***
function GetMemLineColourBlue(id)
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    colourid = GetMemblockInt(id,8)
    result = GetMemColourBlue(colourid)
endfunction result

rem *****
rem *** Place code for PointType here ***
rem *****

rem *****
rem *** Place code for ColourType here ***
rem *****

```

Activity 24.8

Start a new project called *MemblockLine* and implement the code given in FIG-24.23.

Remember to add the functions defined for *PointType* and *ColourType*.

Test and save your project.

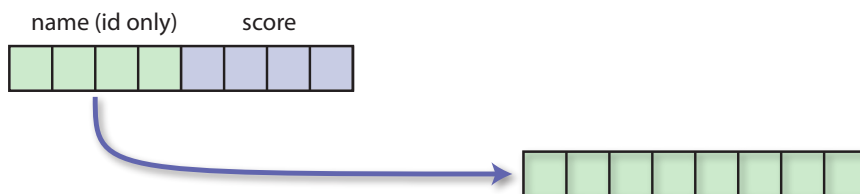
Records Containing Strings

When you need to define one or more fields within a record as a string, use exactly the same approach as that for nested records, with the main memblock containing the ID of the secondary memblock which holds the actual string. For example, FIG-24.24 shows the structure we would use if we wanted to create a record holding the name and high score achieved by the user of a game app.

FIG-24.24

Structure for
HighScoreType

HighScoreType (Internal Structure)



Activity 24.9

Start a new project called *MemblockHighScore* and create the code necessary to allow the creation of a high score record structure as shown in FIG-24.24.

Create the following functions for manipulating the structure:

<code>CreateMemHighScoreType (n\$,hs)</code>	creates a <i>HighScoreType</i> memblock and sets <i>name</i> to <i>n\$</i> and <i>score</i> to <i>hs</i> . Returns the memblock ID.
<code>SetMemHighScoreName (id,n\$)</code>	sets the <i>name</i> field to <i>n\$</i> .
<code>SetMemHighScoreScore (id,hs)</code>	sets the <i>score</i> field to <i>hs</i> .
<code>GetMemHighScoreName (id)</code>	returns the value of <i>name</i> .
<code>GetMemHighScoreScore (id)</code>	returns the value of <i>score</i> .

Parameter *id* is the ID of the memblock.

Make use of the functions previously created in the *MemblockString* project to handle the assignment and retrieval of the player's name.

Test your program by creating two *HighScoreType* variables. The first of these should retain the details given at creation, the second should have the initial name and score details changed. The program should then display the contents of both variables.

Save your project.

Saving Memory Block Data to a File

If the contents of a memblock are required after the app which created them has terminated, then you need to save the memblock's data to a file. Since AGK does not yet contain explicit commands to achieve this, we will need to write our own functions.

When the memblock contains all the data that is to be saved, then the process of writing that information to a file is a simple one. We will need five basic functions:

<i>Open file for writing</i> function	this opens a named file for writing.
<i>Open file for reading</i> function	this opens a named file for reading.
<i>Write data to file</i> function	this writes the contents of one memblock to the file.
<i>Read data from file</i> function	this reads the data required to file on memblock.
<i>Close file</i> function	this closes the file being used.

For example, when a memblock is used to store a `ColourType` structure, then the five functions would be coded as:

```
rem *****
rem ***          Colour File Operations          ***
rem *****

rem *** Opens named file for writing ***
function OpenMemColourFileToWrite(filename$,mode)
    rem *** Create the file ***
    fileid = OpenToWrite(filename$,mode)
endfunction fileid

rem *** Opens named file for reading ***
function OpenMemColourFileToRead(filename$)
    rem *** Create the file ***
    fileid = OpenToRead(filename$)
endfunction fileid

rem *** Writes a single colour (r,g, and b) to the file ***
function WriteMemColour(fileid,id)
    rem *** If memblock or file doesn't exist, exit ***
    if GetMemblockExists(id) = 0 or FileIsOpen(fileid) = 0
        exitfunction
    endif
    rem *** Write colour to file ***
    WriteByte(fileid,GetMemColourRed(id))
    WriteByte(fileid,GetMemColourGreen(id))
    WriteByte(fileid,GetMemColourBlue(id))
endfunction

rem *** Reads a single colour (r,g, and b) from the file ***
function ReadMemColour(fileid,id)
    rem *** If file or colour don't exist, exit ***
    if FileIsOpen(fileid)=0 or GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Read colour from file ***
    SetMemColour(id,ReadByte(fileid),ReadByte(fileid),
    ↳ReadByte(fileid))
endfunction

rem *** Closes file ***
function CloseMemColourFile(fileid)
    rem *** If file ID doesn't exist, exit ***
    if FileIsOpen(fileid) = 0
        exitfunction
    endif
    rem *** Close file ***
    CloseFile(fileid)
endfunction
```

Note that the first two functions do nothing more than call the existing AGK statements for opening files. However, by creating new functions for these operations we make the purpose of the operations more explicit.

Activity 24.10

Load your existing project *MemblockColour* and add the functions given above to your code.

To prove the file-handling functions operate correctly, we will create one ColourType variable, write its data to a disk file, then read that data into a second ColourType variable whose value is then displayed.

Modify the main section of your program to read:

```
rem *** Create a ColourType object ***
colour = CreateMemColourType(120,80,58)
rem *** Write colour data to file ***
file = OpenMemColourFileToWrite("TestData.col",0)
WriteMemColour(file,colour)
rem *** Close the file ***
CloseMemColourFile(file)
rem *** Open the file for reading ***
file = OpenMemColourFileToRead("Testdata.col")
rem *** Read colours into a new ColourType object ***
colour2 = CreateMemColourType(0,0,0)
ReadMemColour(file,colour2)
rem *** Display colour info in new object ***
do
    Print("Red   : "+Str(GetMemColourRed(colour2)))
    Print("Green : "+Str(GetMemColourGreen(colour2)))
    Print("Blue  : "+Str(GetMemColourBlue(colour2)))
    Sync()
loop
```

Test and save your program.

Activity 24.11

Load your existing project *MemblockPoint* and create the five functions necessary to write and read *PointType* data to/from a file. The functions should be named:

```
OpenMemPointFileToWrite()
OpenMemPointFileToRead()
WriteMemPoint()
ReadMemPoint()
CloseMemPointFile()
```

Test your routines by creating a first *PointType* object, writing the object's contents to a file, and then reading the data from the file into a second *PointType* object.

Save your program.

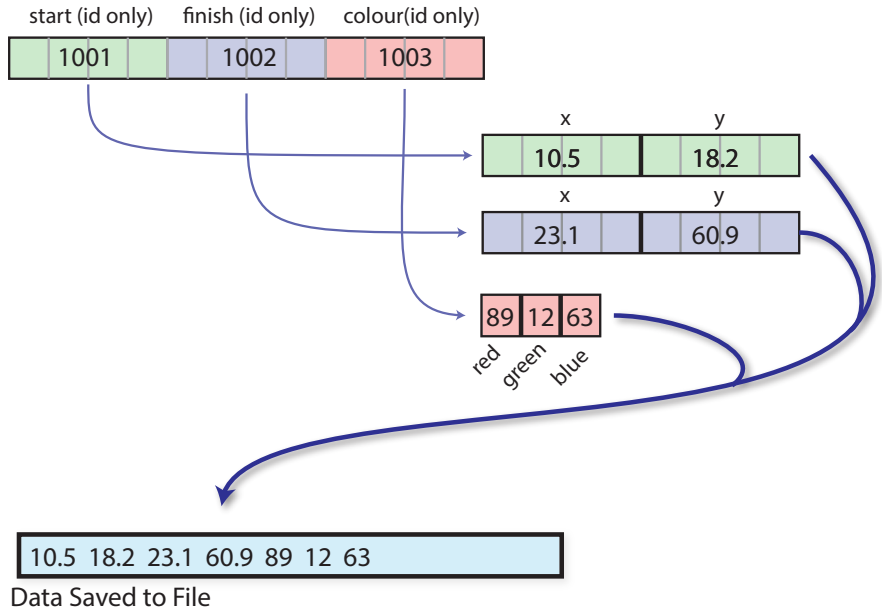
While a record structure such as *ColourType* and *PointType* are easily handled, if the memblock contains the ID's of other memblocks (as we have seen in earlier examples such as *LineType* and *HighScoreType*), then the task of saving the information becomes a little more difficult.

When a record contains the ID of another memblock, saving that ID will achieve nothing since the ID value only has meaning while the program is running. Instead, we need to access the data in that second memblock and write the information we find there to the file (see FIG-24.25).

FIG-24.25

Writing a *LineType* Value to a File

Contents of a LineType Item



Adding save and load features to the *LineType* structure involves the same category of functions as we had in the earlier examples to open the file, close the file, read from the file and write to the file. The code for each of these function is given below.

```
rem *** Opens named file for writing ***
function OpenMemLineFileToWrite(filename$,mode)
    rem *** Create the file ***
    fileid = OpenToWrite(filename$,mode)
    Sleep(1000)
endfunction fileid

rem *** Opens named file for reading ***
function OpenMemLineFileToRead(filename$)
    rem *** Create the file ***
    fileid = OpenToRead(filename$)
endfunction fileid

rem *** Writes a single line's data to the file ***
function WriteMemLine(fileid,id)
    rem *** If memblock or file doesn't exist, exit ***
    if GetMemblockExists(id) = 0 or FileIsOpen(fileid) = 0
        exitfunction
    endif
    rem *** Write data to file ***
    WriteMemPoint(fileid,GetMemblockInt(id,0)) //start
    WriteMemPoint(fileid,GetMemblockInt(id,4)) //finish
    WriteMemColour(fileid,GetMemblockInt(id,8)) //colour
endfunction

rem *** Reads a single line's data from the file ***
function ReadMemLine(fileid,id)
    rem *** If file or colour don't exist, exit ***
```

```

    if FileIsOpen(fileid)=0 or GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Read line data ***
    SetMemLineStart(id,ReadFloat(fileid),ReadFloat(fileid))
    ↪//start
    SetMemLineFinish(id,ReadFloat(fileid),ReadFloat(fileid))
    ↪//finish
    SetLineColour(id,ReadByte(fileid),ReadByte(fileid),
    ↪ReadByte(fileid)) //colour
endfunction

rem *** Closes file ***
function CloseMemLineFile(fileid)
    rem *** If file ID doesn't exist, exit ***
    if FileIsOpen(fileid) = 0
        exitfunction
    endif
    rem *** Close file ***
    CloseFile(fileid)
endfunction

```

While the code within the *open* and *close* functions are identical to previous ones, notice that `WriteMemLine()` makes use of the previously defined `WriteMemPoint()` and `WriteMemColour()` functions to write the three elements that make up a `LineType` object.

Activity 24.12

Load your existing project *MemblockLine* and create the five functions necessary to write and read *LineType* data to/from a file.

Add the file handling functions for *PointType* and *ColourType* at appropriate points in your code.

Test your routines by creating a first *LineType* object whose initial value is [(12,45),(76,80),(128,45,200)] then writing the object's contents to a file.

A second *LineType* object should then be assigned a value by reading the data stored in the file. To check that the program works correctly, ensure that the contents of the second *LineType* object matches that of the first.

Save your program.

Summary

- A memblock is a block of user-allocated memory.
- Use `CreateMemblock()` to allocate a memblock of a specified size.
- Each memblock is assigned a unique ID.
- Locations within a memblock are identified by their offset from the start of the memblock.
- The first location within a memblock has an offset value of zero.
- Use `SetMemblockByte()` to write one byte of data to a single byte location within a specified memblock.

- Use `SetMemblockShort()` to write two bytes of data to a double-byte location within a specified memblock. The data must be stored starting at an even offset value.
- Use `SetMemblockInt()` to write a four-byte integer value to a four-byte location in a specified memblock. The data must be stored using an offset value which is a multiple of four.
- Use `SetMemblockFloat()` to write a four-byte real value to a four-byte location in a specified memblock. The data must be stored using an offset value which is a multiple of four.
- Integer values are written with the least significant byte stored in the first location.
- Use `GetMemblockByte()` to retrieve a single byte of data from a specified memblock. The data is treated as an integer value.
- Use `GetMemblockShort()` to retrieve two bytes of data from a specified memblock. The data is treated as an integer value.
- Use `GetMemblockInt()` to retrieve four bytes of data from a specified memblock. The data is treated as an integer value.
- Use `GetMemblockFloat()` to retrieve four bytes of data from a specified memblock. The data is treated as a real value.
- Use `GetMemblockExists()` to check if a memblock of a specified ID currently exists.
- Use `DeleteMemblock()` to delete an existing memblock.
- Use `GetMemblockSize()` to determine the number of bytes allocated to an existing memblock.
- To store a character in a memblock, use `SetMemblockByte()` after converting the character to an integer using `Asc()`.
- To read a character from a memblock, use `GetMemblockByte()` then convert the value read to a character using `Chr()`.
- A memblock can be used to create data structures which are not allowed when using the `type` statement.
- Using a memblock allows you to update complex data structures from within functions.

Memory Blocks For Images

Introduction

A second use for memory blocks is to use them to hold images. With an image’s data available for manipulation, you can process the image in almost any way possible. For example, an image can be changed from colour to black and white, colours can be reversed, parts of the image can be made transparent, an image’s data can be encrypted, or a new image can be constructed from scratch within a memblock.

Memory Block Image Statements

CreateMemblockFromImage()

The data of a loaded image can be placed within a memblock using the `CreateMemblockFromImage()` statement (see FIG-24.26).

FIG-24.26

CreateMemblock
↳FromImage()

Format 1



Format 2



where

id is an integer value giving the ID to be assigned to the memblock.

imgId is an integer value giving the ID of the existing image that is to be copied into the memblock. Sub-images cannot be transferred to a memblock.

Using the first format requires you to supply the ID to be assigned to the memblock; the second format will automatically assign an ID which is returned by the function.

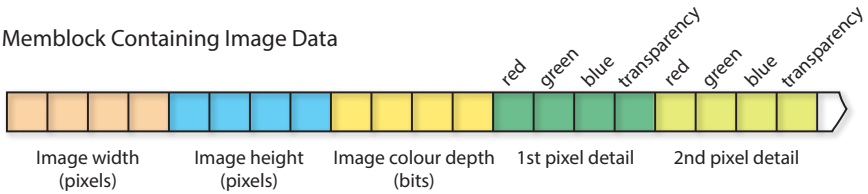
We could load the image data for the file *Bottlebrush.png* into a memblock using the statement:

```
myMemImage = CreateMemblockFromImage (LoadImage (
↳"Bottlebrush.png") )
```

A memblock used to hold an image has a specific format. It starts with details of the image’s dimensions and colour depth and this is followed by the colour and transparency settings for each pixel in the image. The exact format of the memblock is shown in FIG-24.27.

FIG-24.27

Memory Block Image
Data Structure



The memblock is automatically sized to accommodate the loaded image's data.

The pixel data is held in row order. So, if an image was 100 pixels wide by 50 pixels high, the first 100 pixel data entries in the memblock would give the colour settings for the top row of the image. Since each pixel data occupies four bytes, this means that 400 bytes is required to store the first row of the image. With 50 rows, the image would require 400 x 50 bytes + the 12 bytes at the start of the memblock which holds the width, height and colour depth. In other words, the image's memblock requires 2012 bytes (almost 2kB).

The program in FIG-24.28 loads an image and displays the image's width and height.

FIG-24.28

Displaying a Memory
Block Image's
Dimensions

```
rem *** Image Size from Memblock ***

rem *** Load image into memblock ***
myMemImage = CreateMemblockFromImage( loadImage (
    ↪ "Bottlebrush.png" ) )

rem *** Get image dimensions ***
imgwidth = GetMemblockInt( myMemImage, 0 )
imgheight = GetMemblockInt( myMemImage, 4 )

rem *** Display image dimensions ***
do
    Print( "Image Details" )
    Print( "Width   : "+Str( imgwidth ) )
    Print( "Height  : "+Str( imgheight ) )
    Sync()
loop
```

Activity 24.13

Start a new project called *MemblockImage01* and implement the code given in FIG-24.28. Remember to copy *Bottlebrush.png* to the project's *media* folder.

Test and save your project.

Once we have the image's data, you are free to manipulate it in any way you can imagine. For example, we could change the image data from colour to black and white by changing the green and blue values for each pixel to the same as that for the red component.

CreateImageFromMemBlock()

Once the memblock's contents have been changed, you need to create a new version of the actual image. This is done using the `CreateImageFromMemBlock()` statement (see FIG-24.29).

FIG-24.29

CreateImageFrom
↪ Memblock()

Format 1

CreateImageFromMemblock ((id , imemId))

Format 2

integer CreateImageFromMemblock ((imemId))

where

id is an integer value giving the ID to be assigned to the image.

imemId is an integer value giving the ID of the existing memblock that is to be converted to an image.

Using the first format requires you to supply the ID to be assigned to the new image; the second format will automatically assign an ID to the image and return the value of that ID.

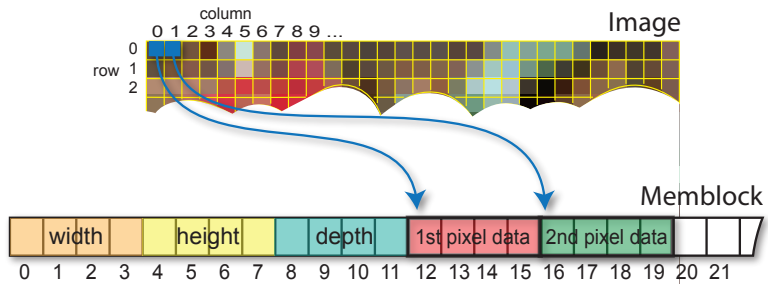
Mapping a Pixel to a Memory Block

If we want to manipulate an image held in a memblock, then we need to be able to convert a pixel's position in the image to the offset value in the memblock where the data for that pixel is stored.

We know that the first 12 bytes of the memblock hold width, height and colour depth information, so the data for the first pixel (top-left pixel) will begin at offset 12 and occupy four bytes (see FIG-24.30).

FIG-24.30

Mapping a Pixel Data to a Position in a Memblock



So, in the first row of pixels (row zero), the pixels' offsets within the memblock are:

pixel 0 = 12
pixel 1 = 16
pixel 2 = 20 ...

Or, in algebraic form:

$$\text{offset} = \text{column} * 4 + 12$$

If the image was 200 pixels wide, then the last pixel of row zero would have its data stored starting at offset

$$\begin{aligned} &= 199 * 4 + 12 \\ &= 808 \end{aligned}$$

With a 200 pixel wide image, 800 bytes are required to store the information about a single row of pixels (4 bytes per pixel). So to calculate the offset for any pixel in a 200 pixel-wide image we would use the formula:

$$\text{offset} = \text{row} * 800 + \text{column} * 4 + 12$$

We can generalise this further for an image of any width as

$$\text{offset} = \text{row} * \text{width} * 4 + \text{column} * 4 + 12$$

Activity 24.14

If an image is 350 pixels wide and 200 pixels high, at which offset would the data for the pixel at row 3 column 12 be stored in a memblock? Assume rows and columns are numbered from zero.

Modifying an Image's Data

The program in FIG-24.31 loads an existing image then converts all the green and blue components of each pixel to match the existing red component's value before reshewing the image. The overall effect of this is to create a greyscale image based on the original image's red values.

FIG-24.31

Changing an Image from
Colour to Black and
White

```
rem *** Images in Memblocks ***

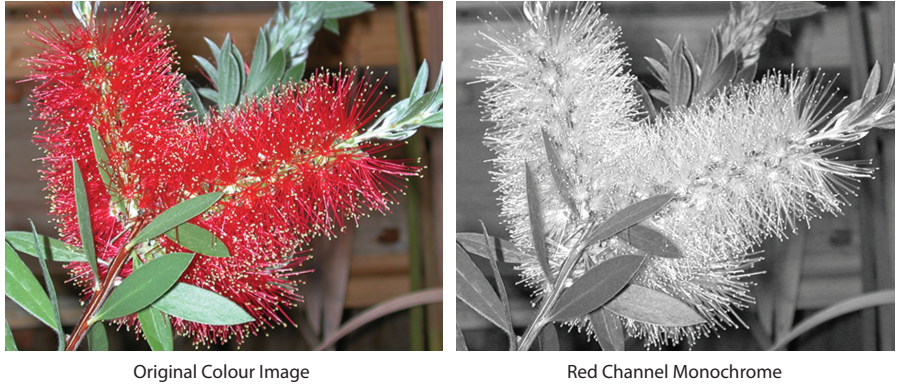
rem *** Load image into memblock ***
myMemImage = CreateMemblockFromImage(LoadImage(
    ↵ "BottleBrush.png"))
rem *** Create Sprite to display image ***
CreateSprite(1,CreateImageFromMemBlock(myMemImage))
SetSpriteSize(1,100,-1)
Sync()
rem *** Start timer ***
ResetTimer
rem *** Not yet switched to monochrome ***
switched = 0
rem *** Convert image to B&W (but don't show it) ***
MonoMemImage(myMemImage)
do
    rem *** After first 2 seconds, show monochrome image ***
    if Timer() > 2 and switched = 0
        SetSpriteImage(1,CreateImageFromMemBlock(myMemImage))
        Sync()
        switched = 1
    endif
loop

rem *** Change to monochrome based on pixel's red value ***
function MonoMemImage(id)
    rem *** If memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Get image dimesions ***
    imgwidth = GetMemblockInt(id,0)
    imgheight = GetMemblockInt(id,4)
    rem *** FOR each pixel... ***
    for row = 0 to imgheight-1
        for col = 0 to imgwidth-1
            rem *** Calculate offset to red value ***
            offset = row*imgwidth*4+col*4+12
            rem *** Get the value of red ***
            redvalue = GetMemblockByte(id,offset)
            rem *** Set green and blue to same value ***
            SetMemblockByte(id,offset+1,redvalue)
            SetMemblockByte(id,offset+2,redvalue)
        next col
    next row
endfunction
```

The image before and after grey-scaling is shown in FIG-24.32.

FIG-24.32

Image in Colour and its
Red Channel



Notice that areas of bright red in the original picture become near-white in the monochrome image, while areas that had little red in the original are near-black in the new image.

Activity 24.15

Start a new project called *MemblockImage02* and implement the code given in FIG-24.31. Remember to copy *Bottlebrush.png* to the project's *media* folder. Test your program.

Modify the function *MonoMemImage()* so that it takes a second integer parameter called *col*. This second parameter should be used to determine how the monochrome image is computed using the following rules:

col	Image processing
0	Based on red value
1	Based on green value
2	Based on blue value
3	Based on average of red, green and blue

Modify the main section of the program so that all five versions of the image are displayed at the same time.

Test and save your project.

Creating Your Own Images from Scratch

Rather than load and manipulate an existing image, you can also create an image from scratch by writing the appropriate value to a memblock and then converting the memblock to an image.

To do this we start by deciding on the dimensions of the image to be constructed and the number of bytes required to store the image. For example, if we want to create an image which is 400 pixels wide and 200 pixels high, then, since each pixel's data requires four bytes (to store colour and transparency details) the complete image would require

$$\begin{aligned} & 400 \times 200 \times 4 \text{ bytes} \\ = & 320,000 \text{ bytes} \end{aligned}$$

As we already know, any memblock which is to be converted to an image must also contain width, height and colour depth information in the first 12 bytes, so the total number of bytes needed in the memblock is:

320,012 bytes

Any function we write to create the required memblock needs to be supplied with the dimensions of the image. These are written to the first 8 bytes of the block (width followed by height). The next 4 bytes are used to contain the colour depth (which is always 32). The function must also return the ID assigned to the new memblock. The code for this function is:

```
rem *** Creates memblock for image ***
function CreateMemImageType(width, height)
  rem *** If dimensions invalid, create 200 by 200
  if width < 1 or width > 4000 or height < 1 or height > 4000
    width = 200
    height = 200
  endif
  rem *** Create memblock for image ***
  id = CreateMemblock(width*height*4 + 12)
  rem *** Store width, height and colour depth ***
  SetMemblockInt(id,0,width)
  SetMemblockInt(id,4,height)
  SetMemblockInt(id,8,32)
endfunction id
```

Notice that the function begins by checking that the dimensions are valid and creates a default 200 by 200 image block if they are not.

A second function is required to write data to the image part of the memblock. Of course, the code for this function will depend on exactly what it is you are trying to create within the image. This is most likely to be the graph of some mathematical function since that is the easiest option to program.

The program in FIG-24.33 uses a draw function which creates a single randomly-positioned pixel in a random colour within the memblock being used to construct an image.

FIG-24.33

Creating An Image
Within a Memblock

```
rem *** Create Image in Memblock ***

rem *** Create memblock for image ***
memid = CreateMemImageType(800,800)

rem *** Create randomly coloured an positioned spots ***
for c = 1 to 40000
  DrawRandomSpotOnMemImage(memid)
next c

rem *** Convert memblock to image ***
imgId = CreateImageFromMemblock(memid)

rem *** Display image in sprite ***
CreateSprite(1,imgId)
SetSpriteSize(1,100,-1)
do
  Sync()
loop
```



FIG-24.33

(continued)

Creating An Image
Within a Memblock

```

rem *** Creates memblock for image ***
function CreateMemImageType(width, height)
    rem *** If dimensions invalid, create 200 by 200
    if width < 1 or width > 4000 or height < 1 or height > 4000
        width = 200
        height = 200
    endif
    rem *** Create memblock for image ***
    id = CreateMemblock(width*height*4 + 12)
    rem **8 Store width, height and colour depth ***
    SetMemblockInt(id,0,width)
    SetMemblockInt(id,4,height)
    SetMemblockInt(id,8,32)
endfunction id

rem *** Places a randomly positioned and coloured pixel ***
function DrawRandomSpotOnMemImage(id)
    rem *** If memblock does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Get dimensions of image ***
    width = GetMemblockInt(id,0)
    height = GetMemblockInt(id,4)
    rem *** Choose random point ***
    rem ** 0 to width -1      **
    x = Random(0,width-1)
    rem ** 0 to height -1    **
    y = Random(0,height-1)
    rem *** Choose random colour ***
    red = Random(0,255)
    green = Random(0,255)
    blue = Random(0,255)
    rem *** Modify appropriate bytes in memblock ***
    post = y*width*4+x*4+12
    SetMemblockByte(id,post,red)
    SetMemblockByte(id,post+1,green)
    SetMemblockByte(id,post+2,blue)
    SetMemblockByte(id,post+3,255)
endfunction

```

Activity 24.16

Start a new project called *MemblockImage* and implement the code given in FIG-24.33.

Test and save your project.

Summary

- Use `CreateMemblockFromImage()` to convert an image's data to a memblock.
- The memblock created from an image uses the first 12 bytes to store the following information:

image width (pixels)	(4 bytes)
image height (pixels)	(4 bytes)
image colour depth	(4 bytes) always set to 32

- An image memblock contains the following information for each pixel in the image:
 - red component (1 byte)
 - green component (1 byte)
 - blue component (1 byte)
 - transparency (1 byte) 0-invisible 255-opaque
- Use `CreateImageFromMemblock()` to convert a memblock of the correct format to an image.
- Modifying the pixel data values will affect the appearance of the image when the memblock is converted back to an image.

Creating a Mandelbrot Image

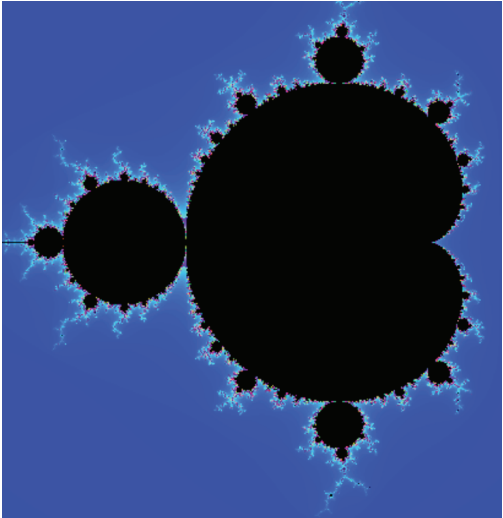
Introduction

FIG-24.34 shows a typical Mandelbrot Set image.

FIG-24.34

The Mandelbrot Set

The Mandelbrot Set

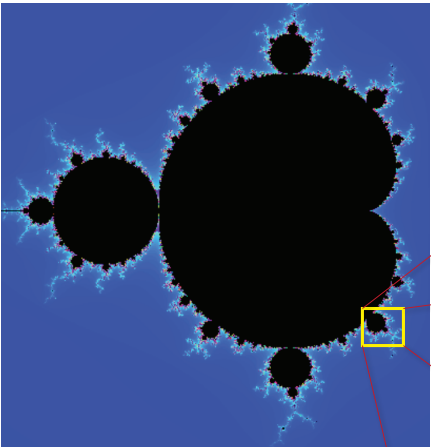


Zooming in on separate areas of the image creates new colourful graphics (see FIG-24.35).

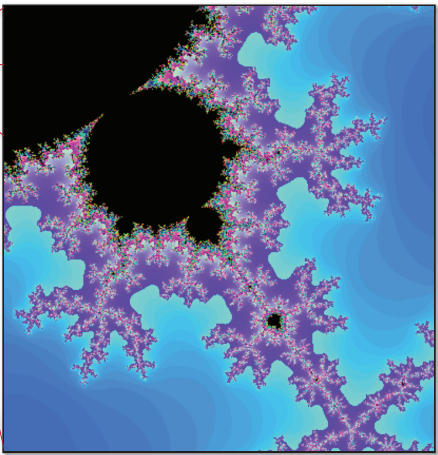
FIG-24.35

Zooming in on the
Mandelbrot Set

The Mandelbrot Set



Magnified Area



What is the Mandelbrot Set?

The Mandelbrot image is created by the application of a mathematical equation and colouring the pixels according to the values derived by that equation.

For a deeper explanation we must start with the definition of imaginary numbers.

The square root of value a is a number, b , such that

$$b \times b = a$$

For example, the square root of 9 is 3 since $3 \times 3 = 9$

But there are some numbers that don't really have a square root which we can write in the traditional way. For example, what is the square root of -9? It isn't -3 since

$$-3 \times -3 = 9$$

To handle such apparently impossible numbers we create a new set of numbers called **imaginary numbers**.

Let's take another look at trying to find the square root of -9:

$$\begin{aligned} & \sqrt{-9} \\ &= \sqrt{9} \times \sqrt{-1} \\ &= 3 \sqrt{-1} \end{aligned}$$

When using imaginary numbers, we use the term i in place of $\sqrt{-1}$. So, $\sqrt{-9}$ is written as

$$3i$$

In general, we end up with an imaginary number when we attempt to find the square root of a negative value.

Activity 24.17

Write down the square root of -25 as an imaginary number.

A **complex number** is one which consists of two part. These are a real component and an imaginary component. For example, the value

$$7 + 4i$$

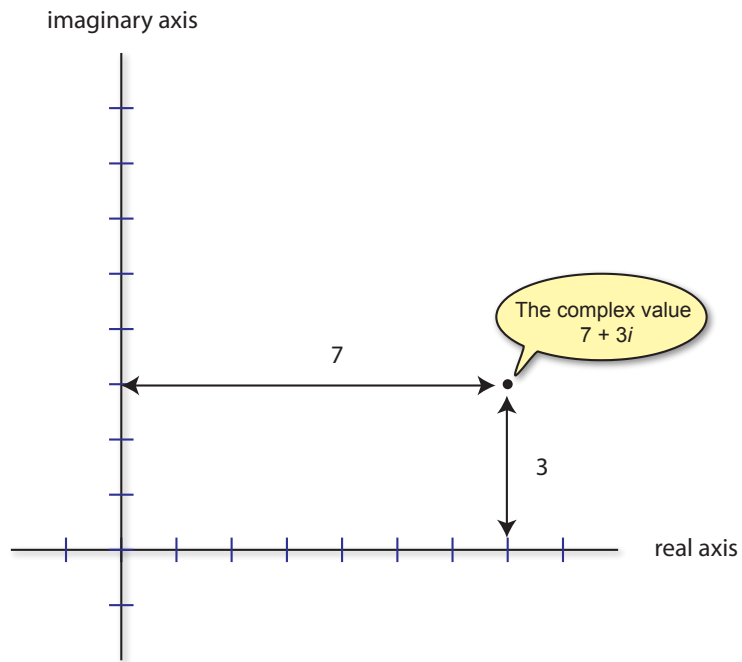
is a complex number.

Perhaps surprisingly, complex numbers are used extensively in many branches of science and engineering.

We can plot complex numbers on a two-dimensional plane if we replace the traditional x-axis with an axis representing the real part of a complex number and replace the y-axis with the imaginary part of the complex number. For example, the value $7 + 4i$ is plotted on the graph shown in FIG-24.36.

FIG-24.36

The Complex Plane



The Mandelbrot series is derived from finding how many iterations can be achieved of the equation

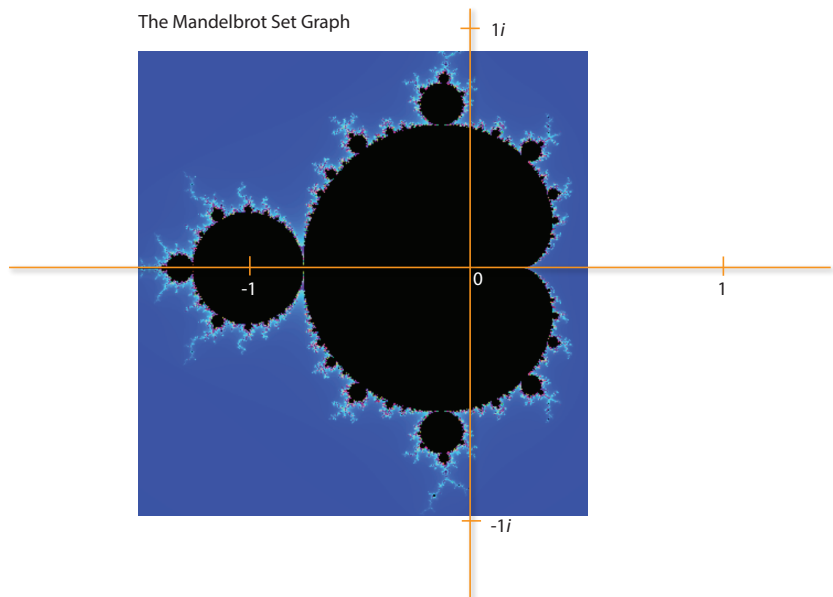
$$Z_{n+1} = Z_n^2 + C$$

before the square of the real component of Z becomes greater than 4. This number determines the colour used for that point on the graph.

The initial value of Z (Z_0) is always zero, but C represents an imaginary number. To create the Mandelbrot graph, the real part of C should lie in the range -2.4 and $+0.8$ and the imaginary part between $-1.2i$ and $+1.2i$ (see FIG-24.37).

FIG-24.37

The Area of the Mandelbrot Set



If, starting with $C = -1 + 0.5i$, it required 125 iterations before the square of the real component of Z was greater than 4, then the point $(-1, 0.5i)$ on the graph would be assigned a colour based on the value 125.

Producing the Program

Don't worry if you didn't follow the math behind the Mandelbrot image (in any case, it was only a brief outline); after all, we are only interested in creating and manipulating the image produced.

To create an image covering part of the Mandelbrot set we need to calculate all combinations of the real and imaginary parts that make up the value C and run them through the equation

$$Z_{n+1} = Z_n^2 + C$$

To this end we need to record the lowest real component and the lowest imaginary component and the range we intend to cover. For example, if we wish to cover the rectangular area from

$$-1.5 - 1.5i \text{ to } 1.0 + 1i$$

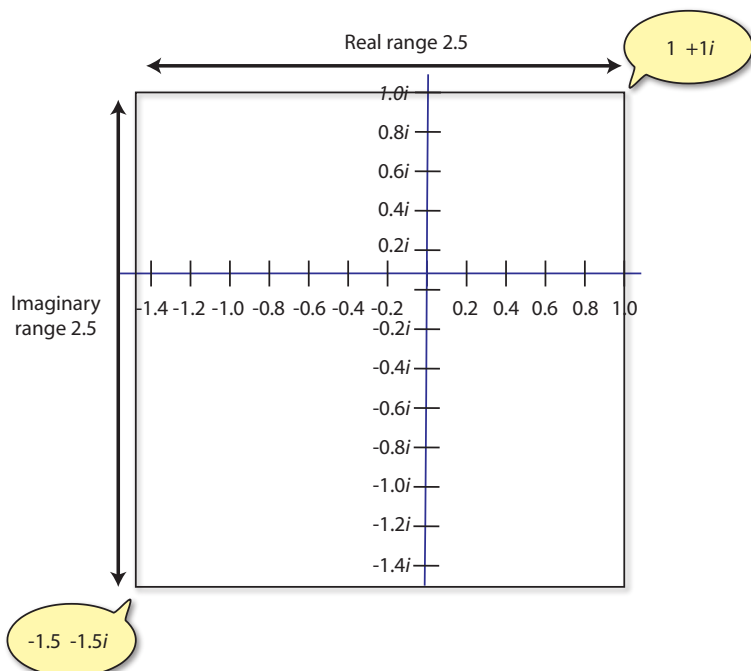
then

$$\begin{aligned} \text{Real}_{\text{start}} &= -1.5 \\ \text{Imaginary}_{\text{start}} &= -1.5 \\ \text{Real}_{\text{range}} &= 2.5 \\ \text{Imaginary}_{\text{range}} &= 2.5 \end{aligned}$$

The area to be drawn is shown in FIG-24.38.

FIG-24.38

Mandelbrot Data



On the computer screen, where the y-axis is flipped over relative to the one in traditional mathematical representation, the point $(-1.5, -1.5i)$ would represent the top-left corner of the area and $(1, 1i)$ the bottom-right.

We'll start by creating a memblock which is designed to contain, not the Mandelbrot image itself, but the start and range information as well as the maximum iterations to be performed when calculating Z and the ID of the second memblock which will contain the actual image. The structure created is equivalent to

```
type MandelbrotDataType
  Rstart#,Istart#    //Starting point of graph
  Rrange#            //Range in real direction
  Irange#            //Range in imaginary direction
  MaxIterations      //Max iterations when calculating Z
  imageID            //ID of memblock containing image
endtype
```

The code for the various routines to create and manipulate this structure is given in FIG-24.39.

FIG-24.39

Creating the Mandelbrot Set

```
rem *** Creates MandelbrotDataType structure and assigns initial
values ***
function CreateMemMandelbrotDataType(Rstart#, Istart#, Rrange#,
Irange#,maxiterations,width,height)
  rem *** Adjust parameters if not sensible ***
  if Rstart# < -2.4
    Rstart# = -2.4
  endif
  if Rstart# > 0.8
    Rstart# = 0.8
  endif
  if Istart# < -1.2
    Istart# = -1.2
  endif
  if Istart# > 1.2
    Istart# = 1.2
  endif
  if Rrange# <= 0 or Rrange# > 3.2
    Rrange# = 3.2
  endif
  if Irange# <= 0 or Irange# > 2.5
    Irange# = 2.5
  endif
  if maxiterations < 16
    maxiterations = 16
  endif
  rem *** Create space for values and blank ID ***
  id = CreateMemblock(24)
  rem *** Set values in memblock ***
  SetMemblockFloat(id,0,Rstart#)
  SetMemblockFloat(id,4,Istart#)
  SetMemblockFloat(id,8,Rrange#)
  SetMemblockFloat(id,12,Irange#)
  SetMemblockInt(id,16,maxiterations)
  rem *** Create memblock for image and record its ID ***
  SetMemblockInt(id,20,CreateMemImageType(width, height))
endfunction id

rem *** Sets the minimum real and imaginary values ***
function SetMemMandelbrotDataStart(id,rs#,is#)
  rem *** If ID does not exist, exit ***
  if GetMemblockExists(id) = 0
    exitfunction
  endif
```




FIG-24.39

(continued)

Creating the Mandelbrot
Set

```

rem *** If invalid start, exit ***
if rs# < -2.4 or rs# > 0.8 or is# < -1.2 or is# > 1.2
    exitfunction
endif
rem *** Set values ***
SetMemblockFloat(id,0,rs#)
SetMemblockFloat(id,4,is#)
endfunction

rem *** Sets the real and imaginary range ***
function SetMemMandelbrotDataRange(id,rr#,ir#)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** If invalid ranges, exit ***
    if rr# <= 0 or rr# > 3.2 or rs# <= 0 or rs# > 2.4
        exitfunction
    endif
    rem *** Set values ***
    SetMemblockFloat(id,8,rr#)
    SetMemblockFloat(id,12,ir#)
endfunction

rem *** Sets max iterations during calculation of Z ***
function SetMemMandelbrotDataMaxIterations(id,max)
    rem *** rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** If parameter too low, exit ***
    if max < 16
        exitfunction
    endif
    rem *** Set maximumiterations ***
    SetMemblockInt(id,16,max)
endfunction

rem *** Returns the minimum real value ***
function GetMemMandelbrotDataStartReal(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Get result ***
    result# = GetMemblockFloat(id,0)
endfunction result#

rem *** Returns the minimum imaginary value ***
function GetMemMandelbrotDataStartImaginary(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Get result ***
    result# = GetMemblockFloat(id,4)
endfunction result#

```



FIG-24.39

(continued)

Creating the Mandelbrot
Set

```

rem *** Returns the range along the real axis ***
function GetMemMandelbrotDataRangeReal(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Get result ***
    result# = GetMemblockFloat(id,8)
endfunction result#

rem *** Returns the range along the imaginary axis ***
function GetMemMandelbrotDataRangeImaginary(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Get result ***
    result# = GetMemblockFloat(id,12)
endfunction result#

rem *** Returns the maximum iterations ***
function GetMemMandelbrotDataMaxIterations(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get result ***
    result = GetMemblockInt(id,16)
endfunction result

rem *** Returns the ID of the Mandelbrot image ***
function GetMemMandelbrotDataImageID(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get result ***
    result = GetMemblockInt(id,20)
endfunction result

rem *** Returns width of image ***
function GetMemMandelbrotDataImageWidth(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get result ***
    result = GetMemblockInt(GetMemMandelbrotDataImageID(id),0)
endfunction result

rem *** Returns height of image ***
function GetMemMandelbrotDataImageHeight(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get result ***
    result = GetMemblockInt(GetMemMandelbrotDataImageID(id),4)
endfunction result

```

Activity 24.18

Start a new project called *Mandelbrot* and implement the code given in FIG-24.39.

Also add the code for *CreateMemImageType()* function which you used in Activity 24.16.

Add a main section to the program and create a **MandelbrotDataType** structure using the following values:

RealStart	-1.7	ImaginaryStart	-1.2
RealRange	2.4	ImaginaryRange	2.4
MaxIterations	256		
ImageWidth	80	ImageHeight	80

Use the *Get* functions to display all of the values held in the new data structure.

ProduceMemMandelbrotImage()

The final function of the basic program is the most complicated one – calculating values of Z for various starting values of C . Each selected value of C corresponds to a pixel. The colour of that pixel is determined by the number of iterations required for Z_{real}^2 to be greater than 4.

The logic of the routine can be expressed as:

```
Retrieve necessary details from Mandelbrot memblock
FOR each pixel column DO
  Map pixel's column value to position on Real axis
  FOR each pixel row DO
    Map pixel's row value to position on Imaginary axis
    Use these real and imaginary values obtained as value of C
    Set  $Z_0$  to 0
    REPEAT
      Calculate new value of  $Z$ 
    UNTIL  $Z_{\text{real}}^2 > 4$  or maximum iterations complete
    Colour the pixel at (row, col) based on number of iterations performed
  ENDFOR
ENDFOR
```

The actual code required to achieve this is shown below:

```
function ProduceMemMandelbrotImage(id)
  rem *** If memblock does not exist, exit ***
  if GetMemblockExists(id) = 0
    exitfunction
  endif
  rem *** Retrieve ID of image area ***
  id1 = GetMemMandelbrotDataImageID(id)
  rem *** Retrieve graph details ***
  rem ** Retrieve image width and height **
  imagewidth = GetMemMandelbrotDataImageWidth(id)
  imageheight = GetMemMandelbrotDataImageHeight(id)
  rem ** Retrieve real and imaginary starts **
  Rstart# = GetMemMandelbrotDataStartReal(id)
  Istart# = GetMemMandelbrotDataStartImaginary(id)
  rem ** Retrieve real and imaginary ranges **
```

```

Rrange# = GetMemMandelbrotDataRangeReal(id)
Irange# = GetMemMandelbrotDataRangeImaginary(id)
rem ** Retrieve max iterations **
max_iterations = GetMemMandelbrotDataMaxIterations(id)
rem *** For each column ***
for x = 0 to imagewidth-1
rem *** Set real part of C ***
Creal# = x*Rrange#/imagewidth + Rstart#
rem *** FOR each row ***
for y = 0 to imageheight-1
rem *** Set imaginary part of C ***
Cimg# = y*Irange#/imageheight + Istart#
rem *** Intialise Z to 0+0i
Zreal# = 0
Zimg# = 0
rem *** Set iteration count to zero ***
count = 0
repeat
inc count
rem *** Calculate next value of Z ***
nextZreal# = Creal# + Zreal#*Zreal# - Zimg#*Zimg#
nextZimg# = Cimg# + 2*Zreal#*Zimg#
Zreal# = nextZReal#
Zimg# = nextZimg#
until (Zreal#^2 + Zimg#^2 > 4) or (count = max_
iterations)
rem *** Set pixel colour ***
r = count*3 mod 256
g = count*8 mod 256
b = 256-count
rem *** Calculate pixel's position in memblock ***
position = x*4+y*imagewidth*4+12
rem *** Write pixel colour to image's memblock ***
SetMemblockByte(id1, position, r)
SetMemblockByte(id1, position+1, g)
SetMemblockByte(id1, position+2, b)
SetMemblockByte(id1, position+3, 255)
next y
next x
endfunction

```

Activity 24.19

Add the code for function `ProduceMemMandelbrotImage()` to the project *Mandelbrot*. Change the main section of the program to read:

```

rem *** Create MandelbrotDataType ***
mandelid = CreateMemMandelbrotDataType(-1.7,-1.2,2.4,2.4,
iterations,256,80,80)
rem *** Create the Mandelbrot image ***
ProduceMemMandelbrotImage(mandelid)
rem *** Create Sprite to display image ***
CreateImageFromMemBlock(1,GetMemMandelbrotDataImageID
iterations,mandelid)
CreateSprite(1,1)
SetSpriteSize(1,100,-1)
do
Sync()
loop

```

Test and save your program.

The image created by Activity 24.19 is very poor quality, but it is created without much delay. Changing the image resolution to 800 by 800 will give a much clearer image but will take some time to produce. Because of this, it is best to display some sort of message to indicate to the user that the program is busy and has not hung up.

Activity 24.20

Modify the main section of *Mandelbrot* so that the image created is 800 by 800 and include a display showing the text *Working...* while the Mandelbrot set is being calculated. The text should be removed when the calculations are complete.

Test and save your program.

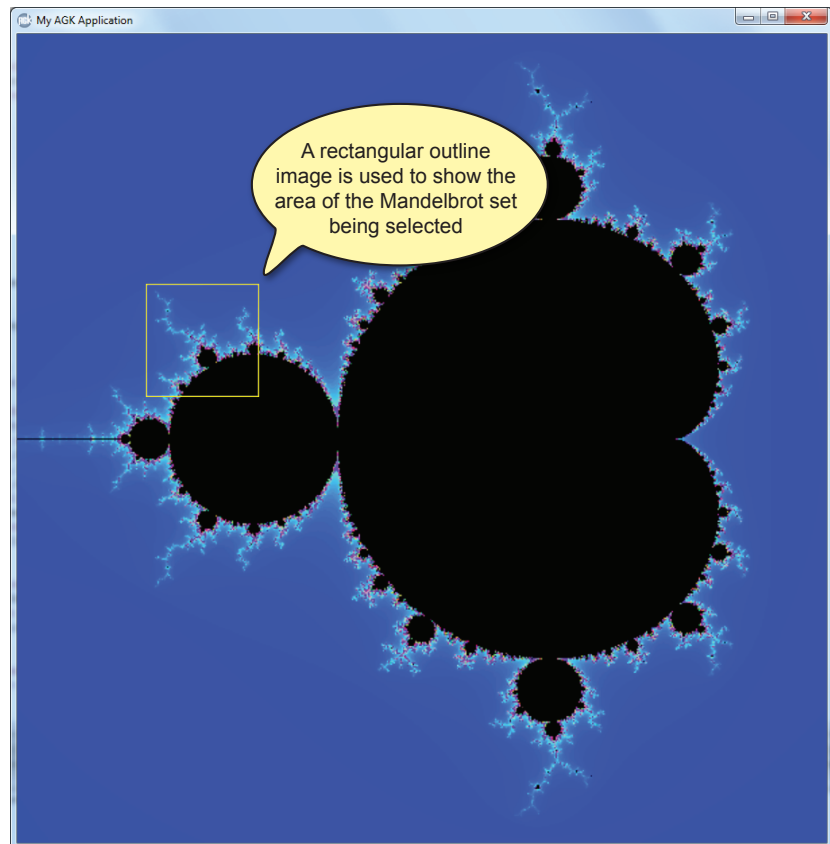
Zooming In

The next feature we need to add to the program is to allow the user to select an area of the image and to recalculate the image to cover only the selected area. This way the user can zoom in on interesting features.

To do this we will make use of an image of a box outline. Clicking on the screen will position the box and then dragging will resize it. This will give the user a visual representation of the area being selected (see FIG-24.40).

FIG-24.40

Selecting an Area of the Mandelbrot Set



The outline box needs to maintain the width-to-height ratio of the Mandelbrot image area, otherwise the next iteration of the image will be distorted.

We need three routines to handle the outline image. The first of these creates and positions the image; the second resizes the image as the pointer is dragged, and the third destroys the image when the next version of the Mandelbrot set is complete. The code for these three routines is given below:

```
rem *** Loads and positions the outline image ***
function CreateBoundary(x#,y#)
    rem *** Load image and display in sprite ***
    LoadImage(33,"Boundary.png")
    CreateSprite(33,33)
    rem *** Position sprite at pointer location
    SetSpritePosition(33,x#,y#)
    rem *** Set sprite to minimum size ***
    SetSpriteSize(33,1,-1)
endfunction

rem *** Resize outline ***
function ResizeBoundary(id)
    rem *** New width (top-left - pointer's x value) ***
    wh = GetPointerX()- GetSpriteX(33)
    rem *** Resize sprite (width to height ratio retained) ***
    SetSpriteSize(33,wh,wh*GetMemMandelbrotDataImageWidth(id)/
    ↳GetMemMandelbrotDataImageHeight(id))
endfunction

rem *** Delete outline sprite and image ***
function DeleteBoundary()
    DeleteSprite(33)
    DeleteImage(33)
endfunction
```

Notice, that for simplicity, the sprite and image elements are assigned a fixed ID value.

Activity 24.21

Add the three functions given above to the end of your program. Save your program.

HandleNewSelection()

The `HandleNewSelection()` function is designed to let the user select an area of the currently displayed Mandelbrot set and create a new image based on that selected area. The concept behind the routine is shown in FIG-24.41.

FIG-24.41

How a New Area is
Calculated

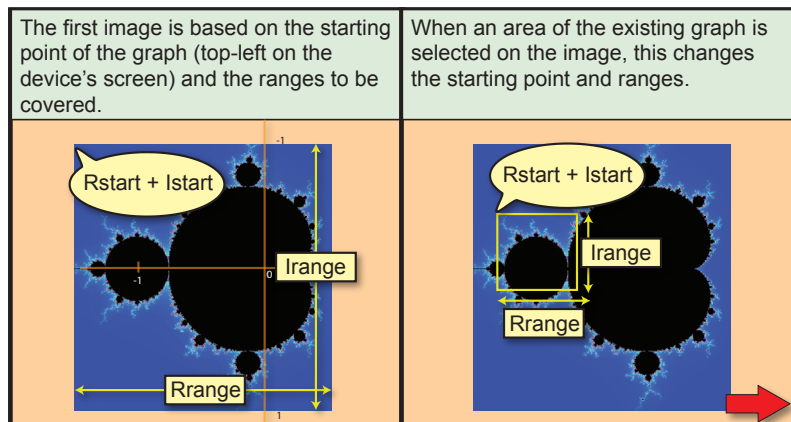
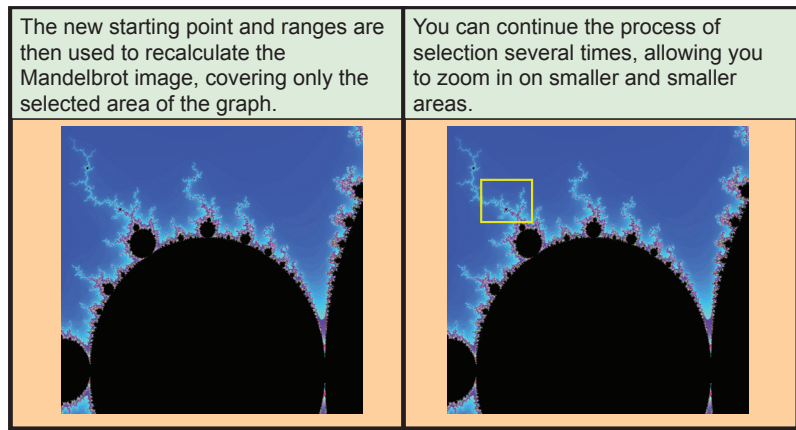


FIG-24.41

(continued)

How a New Area is
Calculated

The logic of the `HandleNewSelection()` function can be summarised as:

```

WHILE pointer pressed DO
  IF pointer just pressed THEN
    Get pointer coordinates
    Place minimised boundary image at pointer coordinates
  ELSE
    Resize boundary image so bottom-right at pointer coordinates
  ENDWHILE
  IF pointer just released THEN
    Change the Mandelbrot start point and ranges to match selected area
    Calculate Mandelbrot image for the new area
    Delete the boundary image
  ENDIF

```

The actual code for the function is:

```

rem *** Allows new area to be selected and redraws image ***
function HandleNewSelection(id)
  rem *** While pointer pressed DO ***
  while GetPointerState() = 1
    rem *** If pointer just pressed... ***
    if GetPointerPressed() = 1
      rem *** ... record pointer position ***
      x# = GetPointerX()
      y# = GetPointerY()
      rem *** create boundary image ***
      CreateBoundary(x#,y#)
    else
      rem *** ... ELSE resize boundary image ***
      ResizeBoundary(id)
    endif
    Sync()
  endwhile
  rem *** IF pointer just released ... ***
  if GetPointerReleased() = 1
    rem *** Calculate top-left coordinates of selected area
    as complex number ***
    px#=(GetSpriteXFromWorld(1,x#,y#)+GetSpriteXByOffset(1))/
    100
    py#=(GetSpriteYFromWorld(1,x#,y#)+GetSpriteYByOffset(1))/
    100
    rem *** Get release coords of pointer ***
    brx# = GetPointerX()
    bry# = GetPointerY()
    rem *** Calculate pointer's position within image ***

```

```

p2x# = (GetSpriteXFromWorld(1,brx#,bry#)+
↳GetSpriteXByOffset(1))/100
p2y# = (GetSpriteYFromWorld(1,brx#,bry#)+
↳GetSpriteYByOffset(1))/100
rem *** Retrieve current Mandelbrot start and range
↳details ***
Rstart# = GetMemMandelbrotDataStartReal(id)
Istart# = GetMemMandelbrotDataStartImaginary(id)
Rrange# = GetMemMandelbrotDataRangeReal(id)
Irange# = GetMemMandelbrotDataRangeImaginary(id)
rem *** Update Mandelbrot start coords ***
SetMemMandelbrotDataStart(id,Rrange#*px# +
↳Rstart#,Irange#*py# + Istart#)
rem *** Calculate and save new range ***
Rnewrange# = Rrange#*p2x# - Rrange#*px#
Inewrange# = RnewRange# *
↳GetMemMandelbrotDataImageHeight(id) /
↳GetMemMandelbrotDataImageWidth(id)
SetMemMandelbrotDataRange(id,Rnewrange#,Inewrange#)
rem *** Calculate Mandelbrot for new area ***
SetTextString(1,"Working...")
Sync()
ProduceMemMandelbrotImage(id)
SetTextString(1,"")
DeleteBoundary()
endif
endfunction

```

Activity 24.22

Add the `HandleNewSelection()` function given above to your program.

Change the program's main section's `do..loop` to read:

```

do
    HandleNewSelection(mandelid)
    DeleteImage(1)
    CreateImageFromMemblock(1,
↳GetMemMandelbrotDataImageID(mandelid))
    SetSpriteImage(1,1)
    Sync()
loop

```

Copy the file *Boundary.png* into the project's *media* folder.

Test your program, checking that the area selection option works correctly.

Save your program.

Shortcomings

There are a couple of problems with the program.

The first of these is that the boundary image cannot be seen correctly when the selected area is too small. We can alleviate that problem by adding the line

```
SetGenerateMipMaps(1)
```


to the start of the main section of the program. By using mipmaps, the image will be displayed at smaller sizes (though it can still be difficult to see).

Activity 24.23

Run your Mandelbrot program and observe the smallest boundary box which can be seen.

Add the `SetGenerateMipMaps(1)` statement to the start of the main section of your project.

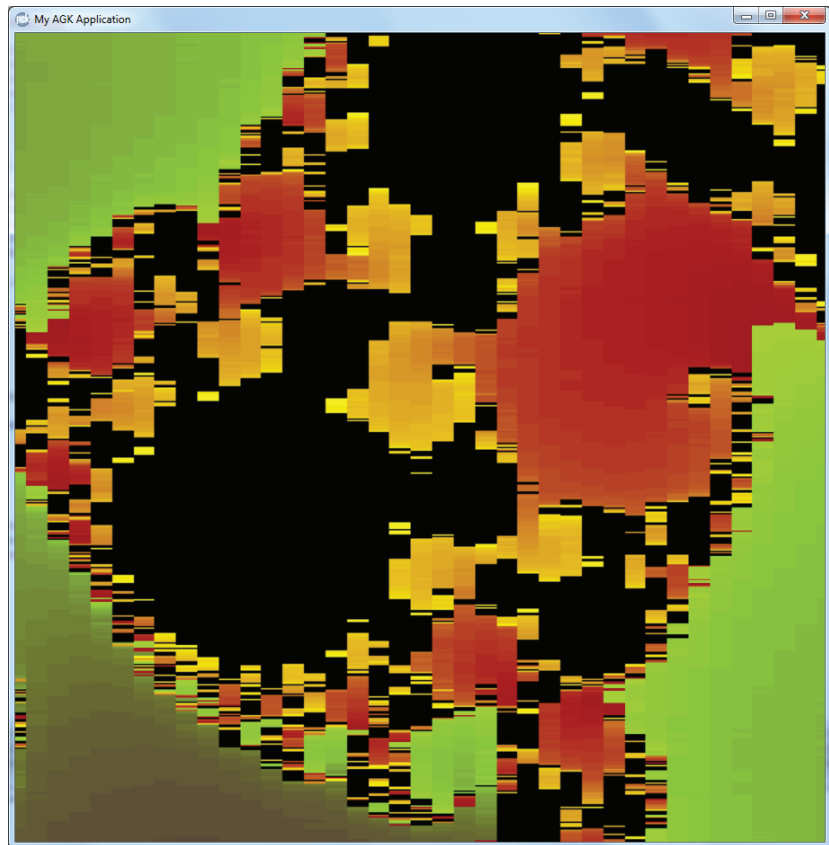
Check how this affects the smallest boundary image visible.

Save your program.

The second problem occurs when you zoom in too far. The image becomes blocky (see FIG-24.42).

FIG-24.42

Zooming in Too Far



This problem is caused by the limited accuracy of the floating point values stored in the program. Remember real values only occupy four bytes and so can only achieve an accuracy to about 6 decimal places. Unfortunately, there is no easy remedy for this problem.

Solutions

Activity 24.1

The second byte is unused because the next value (a short) must start on an even offset byte.

Activity 24.2

No solution required.

Activity 24.3

No solution required.

Activity 24.4

No solution required.

Activity 24.5

To modify the colour, add lines such as those highlighted below to the main section of the program.

```
rem *** Create a memblock of this type ***
mycolour = CreateMemColourType(255,200,100)
rem *** Change colour ***
SetMemColour(mycolour,186,219,88)
rem *** Display the contents of the memblock
do
    Print("Red      : "+Str(GetMemColourRed(mycolour)))
    Print("Green    : " +
    %Str(GetMemColourGreen(mycolour)))
    Print("Blue     : "+Str(GetMemColourBlue(mycolour)))
    Sync()
loop
```

Activity 24.6

Code for *MemblockPoint*:

```
rem *** Point Type as a Memblock ***

remstart
*** Conceptual structure ***
type PointType
    x as float
    y as float
endtype
remend
```

```
rem *** Create required memblock ***
mypoint = CreateMemPointType(23.4,-8.9)
x# = GetMemPointX(mypoint)
y# = GetMemPointY(mypoint)
do
    Print(" (" +Str(x#,2)+", "+Str(y#,2)+") ")
    Sync()
loop
```

```
rem *** Creates and initialises structure ***
rem *** and returns ID ***
function CreateMemPointType(x#,y#)
    rem *** Create memblock ***
    id = CreateMemblock(8)
    rem *** Assign values ***
    SetMemblockFloat(id,0,x#)
    SetMemblockFloat(id,4,y#)
endfunction id
```

```
rem *** Assigns new coordinates ***
function SetMemPoint(id,x#,y#)
    rem *** If memblock doesn't exist, ***
    rem *** exit function ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Assign new x and y values ***
```

```
SetMemblockFloat(id,0,x#)
SetMemblockFloat(id,4,y#)
endfunction

rem *** Get current x value ***
function GetMemPointX(id)
    rem *** If memblock doesn't exist, ***
    rem *** return 0 ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Retrieve current x value ***
    result# = GetMemblockFloat(id,0)
endfunction result#

rem *** Get current x value ***
function GetMemPointY(id)
    rem *** If memblock doesn't exist, ***
    rem *** return 0 ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Retrieve current x value ***
    result# = GetMemblockFloat(id,4)
endfunction result#
```

Activity 24.7

Typical code for adding more weights would be:

```
rem *** Set other weights ***
SetMemRPGWeight(mychar,5,12.3)
SetMemRPGWeight(mychar,10,2.6)
```

The *SetMemRPGWeight()* function does not check that the *idx* parameter is in the range 1 to 10. Such a check could be added using the code:

```
rem *** Set weight in weight[idx] ***
function SetMemRPGWeight(id,idx,w#)
    rem *** If memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** If idx not 1 to 10, exit ***
    if idx < 1 or idx > 10
        exitfunction
    endif
    rem *** Set weight[idx] component ***
    SetMemblockFloat(id,idx*4,w#)
endfunction
```

Activity 24.8

No solution required.

Activity 24.9

The code for *MemblockHighScore*:

```
rem *** Highscore using Memblock ***

rem *** Conceptual Type ***
remstart
type HighScoreType
    name as string
    score as integer
endtype
remend

rem *** Set up both variables ***
myScore1 = CreateMemHighScoreType("Jack Ladd",2890)
myScore2 = CreateMemHighScoreType("",0)
rem *** Change contents of second variable ***
SetMemHighScoreName(myScore2,"Jane Doe")
SetMemHighScoreScore(myScore2,8899)
rem *** Get details from both variables ***
name1$ = GetMemHighScoreName(myScore1)
score1 = GetMemHighScoreScore(myScore1)
name2$ = GetMemHighScoreName(myScore2)
score2 = GetMemHighScoreScore(myScore2)
rem *** Display details of both scores ***
do
    Print("First high score")
    Print("Name      : "+name1$)
    Print("Score     : "+Str(score1))
    Print("Second high score")
```

```

Print("Name : "+name2$)
Print("Score : "+Str(score2))
Sync()
loop
rem *****
rem ***      HighScore Functions      ***
rem *****

function CreateMemHighScoreType(n$,hs)
    rem *** Create memblock for highscore record ***
    id = CreateMemblock(8)
    rem *** Create name in new memblock & store ID ***
    SetMemblockInt(id,0,CreateMemString(n$))
    rem *** Store high score ***
    SetMemblockInt(id,4,hs)
endfunction id

function SetMemHighScoreName(id,n$)
    rem *** If memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Set new name ***
    SetMemString(GetMemblockInt(id,0),n$)
endfunction

function SetMemHighScoreScore(id,hs)
    rem *** If memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Set new high score ***
    SetMemblockInt(id,4,hs)
endfunction

function GetMemHighScoreName(id)
    rem *** If memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction ""
    endif
    result$ = GetMemString(GetMemblockInt(id,0))
endfunction result$

function GetMemHighScoreScore(id)
    rem *** If memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    result = GetMemblockInt(id,4)
endfunction result

rem *****
rem ***      String Functions      ***
rem *****

rem *** Creates memblock for string ***
function CreateMemString(s$)
    rem *** Create memblock for string ***
    id = CreateMemblock(Len(s$))
    rem *** Copy chars from string to memblock ***
    for c = 1 to Len(s$)
        SetMemblockByte(id,c-1,Asc(Mid(s$,c,1)))
    next c
endfunction id

rem *** Set string to s$ ***
function SetMemString(id,s$)
    rem *** If block exists, delete it ***
    if GetMemblockExists(id) = 1
        DeleteMemblock(id)
    endif
    rem *** Create memblock for string ***
    CreateMemblock(id,Len(s$))
    rem *** Copy chars from string to memblock ***
    for c = 1 to Len(s$)
        SetMemblockByte(id,c-1,Asc(Mid(s$,c,1)))
    next c
endfunction

rem *** Retrieve string ***
function GetMemString(id)
    rem *** If no memblock, return empty string ***
    if GetMemblockExists(id) = 0
        exitfunction ""
    endif
    rem *** Start with empty string ***
    result$ = ""
    rem *** Add each character ***

```

```

for c = 0 to GetMemBlockSize(id)-1
    result$ = result$ + Chr(GetMemblockByte(id,c))
next c
endfunction result$

```

Activity 24.10

Modified version of *MemblockColour*:

```

rem *** Colour Type ***

rem *** Conceptual structure ***
remstart
type ColourType
    red
    green
    blue
endtype
remend

rem *** Create a ColourType object ***
colour = CreateMemColourType(120,80,58)
rem *** Write colour data to file ***
file = OpenMemColourFileToWrite("TestData.col",0)
WriteMemColour(file,colour)
rem *** Close the file ***
CloseMemColourFile(file)
rem *** Open the file for reading ***
file = OpenMemColourFileToRead("Testdata.col")
rem *** Read colours to a new ColourType object ***
colour2 = CreateMemColourType(0,0,0)
ReadMemColour(file,colour2)
rem *** Display colour info in new object ***
do
    Print("Red : "+Str(GetMemColourRed(colour2)))
    Print("Green : "+Str(GetMemColourGreen(colour2)))
    Print("Blue : "+Str(GetMemColourBlue(colour2)))
    Sync()
loop

rem *****
rem ***      Basic Colour Operations      ***
rem *****

rem *** Creates and initialises the structure ***
rem *** returns the ID ***
function CreateMemColourType(r,g,b)

    rem *** Create memblock for record ***

    id = CreateMemblock(3)

    rem *** Store initial colours in range 0-255 ***
    SetMemblockByte(id,0,r mod 256)
    SetMemblockByte(id,1,g mod 256)
    SetMemblockByte(id,2,b mod 256)
endfunction id

rem *** Change all three colour elements ***
function SetMemColour(id,r,g,b)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Set red component ***
    SetMemblockByte(id,0,r mod 256)
    rem *** Set green component ***
    SetMemblockByte(id,1,g mod 256)
    rem *** Set blue component ***
    SetMemblockByte(id,2,b mod 256)
endfunction

rem *** Change red only ***
function SetMemColourRed(id,r)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Set red component ***
    SetMemblockByte(id,0,r mod 256)
endfunction

rem *** Change green only ***
function SetMemColourGreen(id,g)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif

```

```

    rem *** Set green component ***
    SetMemblockByte(id,1,g mod 256)
endfunction

rem *** Change blue only ***
function SetMemColourBlue(id,r)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Set blue component ***
    SetMemblockByte(id,2,b mod 256)
endfunction

rem *** Get current red setting ***
function GetMemColourRed(id)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction 0
    endif
    rem *** Retrieve red value ***
    result = GetMemblockByte(id,0)
endfunction result

rem *** Get current green setting ***
function GetMemColourGreen(id)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction 0
    endif
    rem *** Retrieve red value ***
    result = GetMemblockByte(id,1)
endfunction result

rem *** Get current blue setting ***
function GetMemColourBlue(id)
    rem *** if memblock does not exist, exit ***
    if GetMemblockExists(id)=0
        exitfunction 0
    endif
    rem *** Retrieve red value ***
    result = GetMemblockByte(id,2)
endfunction result

rem *****
rem *** Colour File Operations ***
rem *****

rem *** Opens named file for writing ***
function OpenMemColourFileToWrite(filename$,mode)
    rem *** Create the file ***
    fileid = OpenToWrite(filename$,mode)
    Sleep(1000)
endfunction fileid

rem *** Opens named file for reading ***
function OpenMemColourFileToRead(filename$)
    rem *** Create the file ***
    fileid = OpenToRead(filename$)
endfunction fileid

rem *** Writes a single colour (r,g,b) to file ***
function WriteMemColour(fileid,id)
    rem *** If memblock or file doesn't exist, exit ***
    if GetMemblockExists(id)=0 or FileIsOpen(fileid)=0
        exitfunction
    endif
    rem *** Write data to file ***
    WriteByte(fileid,GetMemColourRed(id))
    WriteByte(fileid,GetMemColourGreen(id))
    WriteByte(fileid,GetMemColourBlue(id))
endfunction

rem *** Reads a single colour (r,g,b) from file ***
function ReadMemColour(fileid,id)
    rem *** If file or colour don't exist, exit ***
    if FileIsOpen(fileid)=0 or GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Read colour ***
    SetMemColour(id,ReadByte(fileid),ReadByte(fileid),
    ReadByte(fileid))
endfunction

rem *** Closes file ***
function CloseMemColourFile(fileid)
    rem *** If file ID doesn't exist, exit ***
    if FileIsOpen(fileid) = 0

```

```

        exitfunction
    endif
    rem *** Close file ***
    CloseFile(fileid)
endfunction

```

Activity 24.11

Modified version of *MemblockPoint*:

```

rem *** Point Type as a Memblock ***
remstart
*** Conceptual structure ***
type PointType
    x as float
    y as float
endtype
remend

rem *** Create a PointType object ***
point = CreateMemPointType(12.6,20.9)
rem *** Write point data to file ***
file = OpenMemPointFileToWrite("TestData.pnt",0)
WriteMemPoint(file,point)
rem *** Close the file ***
CloseMemPointFile(file)
rem *** Open the file for reading ***
file = OpenMemPointFileToRead("Testdata.pnt")
rem *** Read colours into new ColourType object ***
point2 = CreateMemPointType(0,0)
ReadMemPoint(file,point2)
rem *** Display colour info in new object ***
do
    Print("Point2 : (" + Str(GetMemPointX(point2),1) +
    " , " + Str(GetMemPointY(point2),1) + ")")
    Sync()
loop

rem *** Creates and initialises structure ***
rem *** and returns ID ***
function CreateMemPointType(x#,y#)
    rem *** Create memblock ***
    id = CreateMemblock(8)
    rem *** Assign values ***
    SetMemblockFloat(id,0,x#)
    SetMemblockFloat(id,4,y#)
endfunction id

rem *** Assigns new coordinates ***
function SetMemPoint(id,x#,y#)
    rem *** If memblock doesn't exist, ***
    rem *** exit function ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Assign new x and y values ***
    SetMemblockFloat(id,0,x#)
    SetMemblockFloat(id,4,y#)
endfunction

rem *** Get current x value ***
function GetMemPointX(id)
    rem *** If memblock doesn't exist, ***
    rem *** return 0 ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Retrieve current x value ***
    result# = GetMemblockFloat(id,0)
endfunction result#

rem *** Get current y value ***
function GetMemPointY(id)
    rem *** If memblock doesn't exist, ***
    rem *** return 0 ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Retrieve current x value ***
    result# = GetMemblockFloat(id,4)
endfunction result#

rem *** Opens named file for writing ***
function OpenMemPointFileToWrite(filename$,mode)
    rem *** Create the file ***
    fileid = OpenToWrite(filename$,mode)

```

```

endfunction fileid

rem *** Opens named file for reading ***
function OpenMemPointFileToRead(filename$)
    rem *** Create the file ***
    fileid = OpenToRead(filename$)
endfunction fileid

rem *** Writes a single point(x,y) to the file ***
function WriteMemPoint(fileid,id)
    rem *** If point or file doesn't exist, exit ***
    if GetMemblockExists(id)=0 or FileIsOpen(fileid)=0
        exitfunction
    endif
    rem *** Write point to file ***
    WriteFloat(fileid,GetMemPointX(id))
    WriteFloat(fileid,GetMemPointY(id))
endfunction

rem *** Reads a single point (x,y) from the file ***
function ReadMemPoint(fileid,id)
    rem *** If file or point don't exist, exit ***
    if FileIsOpen(fileid)=0 or GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Read point ***
    SetMemPoint(id,ReadFloat(fileid),ReadFloat(fileid))
endfunction

rem *** Closes file ***
function CloseMemPointFile(fileid)
    rem *** If file ID doesn't exist, exit ***
    if FileIsOpen(fileid) = 0
        exitfunction
    endif
    rem *** Close file ***
    CloseFile(fileid)
endfunction

```

Activity 24.12

Modified version of *MemblockLine*:

```

rem *** Writing a LineType Item to a File ***

rem *** Create a LineType structure ***
line1 = CreateMemLineType()

rem *** Assign start, finish and colour to line ***
SetMemLineStart(line1,12,45)
SetMemLineFinish(line1,76,80)
SetMemLineColour(line1,128,45,200)

rem *** Write line data to file ***
file = OpenMemLineFileToWrite("TestData.lin",0)
WriteMemLine(file,line1)
rem *** Close the file ***
CloseMemLineFile(file)
rem *** Open the file for reading ***
file = OpenMemLineFileToRead("TestData.lin")
rem *** Read line data into a new LineType object ***
line2 = CreateMemLineType()
ReadMemLine(file,line2)
rem *** Display line info in new object ***
do
    Print("Start : "+Str(GetMemLineStartX(line2),2)
    &"+", " "+Str(GetMemLineStartY(line2),1)+"")
    Print("Finish: "+Str(GetMemLineFinishX(line2),1)
    &"+", " "+Str(GetMemLineFinishY(line2),1)+"")
    Print("Red : "+Str(GetMemLineColourRed(line2)))
    Print("Green : "+Str(GetMemLineColourGreen(
    &line2)))
    Print("Blue : "+Str(GetMemLineColourBlue(line2)))
    Sync()
loop

rem *** Creates a linetype with default values ***
rem *** Start=(0,0); finish = (0,0) ***
rem *** colour = 255,255,255 (white) ***
function CreateMemLineType()
    rem *** Create main memblock ***
    id = CreateMemblock(12)
    rem *** Create memblock for fields in record ***

```

```

    startid = CreateMemPointType(0,0)
    finishid = CreateMemPointType(0,0)
    colourid = CreateMemColourType(255,255,255)
    rem *** Store IDs in main memblock ***
    SetMemblockInt(id,0,startid)
    SetMemblockInt(id,4,finishid)
    SetMemblockInt(id,8,colourid)
endfunction id

rem *** Sets start point to (x#,y#) ***
function SetMemLineStart(id,x#,y#)
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    startid = GetMemblockInt(id,0)
    SetMemPoint(startid,x#,y#)
endfunction

rem *** Sets finish point to (x#,y#) ***
function SetMemLineFinish(id,x#,y#)
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    finishid = GetMemblockInt(id,4)
    SetMemPoint(finishid,x#,y#)
endfunction

rem *** Sets the line's colour to r,g,b ***
function SetMemLineColour(id,r,g,b)
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    colourid = GetMemblockInt(id,8)
    SetMemColour(colourid,r,g,b)
endfunction

rem *** Returns the X coord of the start point ***
function GetMemLineStartX(id)
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    startid = GetMemblockInt(id,0)
    result# = GetMemPointX(startid)
endfunction result#

rem *** Returns the Y coord of the start point ***
function GetMemLineStartY(id)
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    startid = GetMemblockInt(id,0)
    result# = GetMemPointY(startid)
endfunction result#

rem *** Returns the X coord of the finish point ***
function GetMemLineFinishX(id)
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    finishid = GetMemblockInt(id,4)
    result# = GetMemPointX(finishid)
endfunction result#

rem *** Returns the Y coord of the finish point ***
function GetMemLineFinishY(id)
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    finishid = GetMemblockInt(id,4)
    result# = GetMemPointY(finishid)
endfunction result#

rem *** Returns red component of line's colour ***
function GetMemLineColourRed(id)
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    colourid = GetMemblockInt(id,8)
    result = GetMemColourRed(colourid)
endfunction result

rem *** Returns the green component of the line's
colour ***
function GetMemLineColourGreen(id)
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    colourid = GetMemblockInt(id,8)
    result = GetMemColourGreen(colourid)

```

```

endfunction result

rem *** Returns the blue component of the line's
colour ***
function GetMemLineColourBlue(id)
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    colourid = GetMemblockInt(id,8)
    result = GetMemColourBlue(colourid)
endfunction result

rem *** Opens named file for writing ***
function OpenMemLineFileToWrite(filename$,mode)
    rem *** Create the file ***
    fileid = OpenToWrite(filename$,mode)
    Sleep(1000)
endfunction fileid

rem *** Opens named file for reading ***
function OpenMemLineFileToRead(filename$)
    rem *** Create the file ***
    fileid = OpenToRead(filename$)
endfunction fileid

rem *** Writes a single line's data to the file ***
function WriteMemLine(fileid,id)
    rem *** If memblock or file doesn't exist, exit ***
    if GetMemblockExists(id)=0 or FileIsOpen(fileid)=0
        exitfunction
    endif
    rem *** Write data to file ***
    WriteMemPoint(fileid,GetMemblockInt(id,0)) //start
    WriteMemPoint(fileid,GetMemblockInt(id,4)) //finish
    WriteMemColour(fileid,GetMemblockInt(id,8)) //
    colour
endfunction

rem *** Reads a single line's data from the file ***
function ReadMemLine(fileid,id)
    rem *** If file or colour don't exist, exit ***
    if FileIsOpen(fileid)=0 or GetMemblockExists(id)=0
        exitfunction
    endif
    rem *** Read line data ***
    SetMemLineStart(id,ReadFloat(fileid),
    ReadFloat(fileid))//start
    SetMemLineFinish(id,ReadFloat(fileid),
    ReadFloat(fileid))//finish
    SetMemLineColour(id,ReadByte(fileid),
    ReadByte(fileid)) //colour
endfunction

rem *** Closes file ***
function CloseMemLineFile(fileid)
    rem *** If file ID doesn't exist, exit ***
    if FileIsOpen(fileid) = 0
        exitfunction
    endif
    rem *** Close file ***
    CloseFile(fileid)
endfunction

***** Add Code for PointType here *****

***** Add Code for ColourType here *****

```

Activity 24.13

No solution required.

Activity 24.14

If the image is 350 pixels wide, then each row requires

350 x 4 bytes

= 1400 bytes

So rows 0 to 2 require a total of 4200 bytes.

Pixels 0 to 11 of row 3 need 48 bytes

So, remembering the 12 bytes needed at the start of an image memblock, the offset required to access row 3 column 12 is:

$$12 + 4200 + 48 = 4260$$

Activity 24.15

The new version of the function *MonoMemImage()* could be coded as:

```

rem *** Change to monochrome based on selected
option ***

function MonoMemImage(id,opt)
    rem *** Get image dimensions ***
    imgwidth = GetMemblockInt(id,0)
    imgheight = GetMemblockInt(id,4)
    rem *** FOR each pixel... ***
    for row = 0 to imgheight-1
        for col = 0 to imgwidth-1
            select opt
            case 0 // red
                rem *** Calculate offset to red ***
                offset = row*imgwidth*4+col*4+12
                rem *** Get the value of red ***
                redvalue = GetMemblockByte(id,offset)
                rem *** Green & blue to same value ***
                SetMemblockByte(id,offset+1,redvalue)
                SetMemblockByte(id,offset+2,redvalue)
            endcase
            case 1 //green
                rem *** Offset to green value ***
                offset = row*imgwidth*4+col*4+12 + 1
                rem *** Get the value of green ***
                greenvalue = GetMemblockByte(id,offset)
                rem *** Red & blue to same value ***
                SetMemblockByte(id,offset-1,greenvalue)
                SetMemblockByte(id,offset+1,greenvalue)
            endcase
            case 2 //blue
                rem *** Offset to blue value ***
                offset = row*imgwidth*4+col*4+12 + 2
                rem *** Get the value of blue ***
                bluevalue = GetMemblockByte(id,offset)
                rem *** Red & green to same value ***
                SetMemblockByte(id,offset-2,bluevalue)
                SetMemblockByte(id,offset-1,bluevalue)
            endcase
            case 3 //average
                rem *** Offset to red value ***
                offset = row*imgwidth*4+col*4+12
                rem *** Get the value of average ***
                averagevalue =
                (GetMemblockByte(id,offset)+
                GetMemblockByte(id,offset+1)+
                GetMemblockByte(id,offset+2))/3
                rem *** Set set colours to average ***
                SetMemblockByte(id,offset,averagevalue)
                SetMemblockByte(id,offset+1,averagevalue)
                SetMemblockByte(id,offset+2,averagevalue)
            endcase
        endselect
    next col
next row
endfunction

```

To display all five versions of the image at the same time the following code could be used.

```

rem *** Images in memblocks ***

rem *** Load image into memblock ***

myMemImage = CreateMemblockFromImage
    (LoadImage("BottleBrush.png"))
rem *** Show original image in sprite ***
CreateImageFromMemblock(1,myMemImage)
CreateSprite(1,1)
SetSpriteSize(1,48,-1)
rem *** Create remaining four versions of image ***
for c = 2 to 5
    rem *** Convert memblock to B&W ***
    MonoMemImage(myMemImage,c-2)
rem *** Show image in new sprite ***

```

```

CreateImageFromMemblock(c,myMemImage)
CreateSprite(c,c)
SetSpriteSize(c,48,-1)
SetSpritePosition(c,(c-1) mod 2 * 50, (c-1)/
  2 * 33)
rem *** Set up memblock for next conversion ***
DeleteMemblock(myMemblock)
myMemImage = CreateMemblockFromImage
  (LoadImage("BottleBrush.png"))
next c
rem *** Show the five versions ***
do
  Sync()
loop

```

The code was used with a screen setting of 600 x 750.

The line

```
SetSpritePosition(c,(c-1) mod 2 * 50, (c-1)/2 * 33)
```

is used to position two images per row, with three rows (only one image appears on the third row).

Activity 24.16

No solution required.

Activity 24.17

```

√-25
= √25 x √-1
= 5√-1
= 5i

```

Activity 24.18

Code for *Mandelbrot*:

```

rem *** Use MandelbrotDataType ***

rem *** Create MandelbrotDataType item ***
mandelbrot =
  CreateMemMandelbrotDataType(-1.7,-1.2,2.4,
  2.4,256,80,80)
rem *** Display item's data ***
do
  Print("Start          : ( " +
  Str(GetMemMandelbrotDataStartReal(mandelbrot),1)
  " , " + Str(GetMemMandelbrotDataStartImaginary(
  mandelbrot),1)+") ")
  Print("Real range      : " +
  Str(GetMemMandelbrotDataRangeReal(mandelbrot),1))
  Print("Imaginary range : " +
  Str(GetMemMandelbrotDataRangeImaginary(
  mandelbrot),1))
  Print("Maximum Iterations : " +
  Str(GetMemMandelbrotDataMaxIterations(
  mandelbrot)))
  Print("Image ID         : " +
  Str(GetMemMandelbrotDataImageID(mandelbrot)))
  Print("Image width      : " +
  Str(GetMemMandelbrotDataImageWidth(mandelbrot)))
  Print("Image height     : " +
  Str(GetMemMandelbrotDataImageHeight(mandelbrot)))
  Sync()
loop

rem *****
rem *** MandelbrotDataType Functions ***
rem *****

rem *** Creates MandelbrotDataType structure and
assigns initial values ***
function CreateMemMandelbrotDataType(Rstart#,
  Istart#, Rrange#, Irange#,maxiterations,width,
  height)
  rem *** Adjust parameters if not sensible ***
  if Rstart# < -2.4
    Rstart# = -2.4
  endif
  if Rstart# > 0.8
    Rstart# = 0.8
  endif
  if Istart# < -1.2

```

```

  Istart# = -1.2
endif
  if Istart# > 1.2
    Istart# = 1.2
  endif
  if Rrange# <= 0 or Rrange# > 3.2
    Rrange# = 3.2
  endif
  if Irange# <= 0 or Irange# > 2.4
    Irange# = 2.4
  endif
  if maxiterations < 16
    maxiterations = 16
  endif
  rem *** Create space for values and blank ID ***
  id = CreateMemblock(24)
  rem *** Set values in memblock ***
  SetMemblockFloat(id,0,Rstart#)
  SetMemblockFloat(id,4,Istart#)
  SetMemblockFloat(id,8,Rrange#)
  SetMemblockFloat(id,12,Irange#)
  SetMemblockInt(id,16,maxiterations)
  rem *** Create memblock for image and record its
  ID ***
  SetMemblockInt(id,20,CreateMemImageType(width,
  height))
endfunction id

```

```

rem *** Sets minimum real and imaginary values ***
function SetMemMandelbrotDataStart(id,rs#,is#)
  rem *** If ID does not exist, exit ***
  if GetMemblockExists(id) = 0
    exitfunction
  endif
  rem *** If invalid start, exit ***
  if rs# < -2.4 or rs# > 0.8 or is# < -1.2 or
  is# > 1.2
    exitfunction
  endif
  rem *** Set values ***
  SetMemblockFloat(id,0,rs#)
  SetMemblockFloat(id,4,is#)
endfunction

```

```

rem *** Sets the real and imaginary range ***
function SetMemMandelbrotDataRange(id,rr#,ir#)
  rem *** If ID does not exist, exit ***
  if GetMemblockExists(id) = 0
    exitfunction
  endif
  rem *** If invalid ranges, exit ***
  if rr# <= 0 or rr# > 3.2 or ir# <= 0 or ir# > 3.2
    exitfunction
  endif
  rem *** Set values ***
  SetMemblockFloat(id,8,rr#)
  SetMemblockFloat(id,12,ir#)
endfunction

```

```

function SetMemMandelbrotDataMaxIterations(id,max)
  rem *** rem *** If ID does not exist, exit ***
  if GetMemblockExists(id) = 0
    exitfunction
  endif
  rem *** If parameter too low, exit ***
  if max < 16
    exitfunction
  endif
  rem *** Set maximum iterations ***
  SetMemblockInt(id,16,max)
endfunction

```

```

rem *** Returns the minimum real value ***
function GetMemMandelbrotDataStartReal(id)
  rem *** If ID does not exist, exit ***
  if GetMemblockExists(id) = 0
    exitfunction 0.0
  endif
  rem *** Get result ***
  result# = GetMemblockFloat(id,0)
endfunction result#

```

```

rem *** Returns the minimum imaginary value ***
function GetMemMandelbrotDataStartImaginary(id)
  rem *** If ID does not exist, exit ***
  if GetMemblockExists(id) = 0
    exitfunction 0.0
  endif
  rem *** Get result ***
  result# = GetMemblockFloat(id,4)

```



```

endfunction result#

rem *** Returns the range along the real axis ***
function GetMemMandelbrotDataRangeReal(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Get result ***
    result# = GetMemblockFloat(id,8)
endfunction result#

rem *** Returns range along the imaginary axis ***
function GetMemMandelbrotDataRangeImaginary(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Get result ***
    result# = GetMemblockFloat(id,12)
endfunction result#

rem *** Returns the maximum iterations ***
function GetMemMandelbrotDataMaxIterations(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get result ***
    result = GetMemblockInt(id,16)
endfunction result

rem *** Returns the ID of the mandelbrot image ***
function GetMemMandelbrotDataImageID(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get result ***
    result = GetMemblockInt(id,20)
endfunction result

rem *** Returns width of image ***
function GetMemMandelbrotDataImageWidth(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get result ***
    result =
        ↵GetMemblockInt(GetMemMandelbrotDataImageID(id),
        ↵0)
endfunction result

rem *** Returns height of image ***
function GetMemMandelbrotDataImageHeight(id)
    rem *** If ID does not exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get result ***
    result =
        ↵GetMemblockInt(GetMemMandelbrotDataImageID(id),
        ↵4)
endfunction result

rem *****
rem *** ImageType Functions ***
rem *****

rem *** Creates memblock for image ***
function CreateMemImageType(width, height)
    rem *** If dimensions invalid, create 200 by 200
    if width < 1 or width > 4000 or height < 1 or
        ↵height > 4000
        width = 200
        height = 200
    endif
    rem *** Create memblock for image ***
    id = CreateMemblock(width*height*4 + 12)
    rem *** Store width, height and colour depth ***
    SetMemblockInt(id,0,width)
    SetMemblockInt(id,4,height)
    SetMemblockInt(id,8,32)
endfunction id

```

Activity 24.19

No solution required.

Activity 24.20

Modified code for the main section of *Mandelbrot*:

```

rem *** Displaying the Mandelbrot Set ***

rem *** Create MandelbrotDataType ***
mandelid = CreateMemMandelbrotDataType(-1.7,-1.2,2.4,
    ↵2.4,256,800,800)
rem *** Display message ***
CreateText(1,"Working...")
Sync()
rem *** Create the Mandelbrot image ***
ProduceMemMandelbrotImage(mandelid)
rem *** Create Sprite to display image ***
CreateImageFromMemBlock(1,
    ↵GetMemMandelbrotDataImageID(mandelid))
CreateSprite(1,1)
SetSpriteSize(1,100,-1)
rem *** Remove message ***
SetTextString(1,"")
rem *** Display ***
do
    Sync()
loop

```

Activity 24.21

No solution required.

Activity 24.22

Change the `do..loop` structure in the main section of *Mandelbrot* to read:

```

do
    HandleNewSelection(mandelid)
    DeleteImage(1)
    CreateImageFromMemblock(1,
        ↵↵GetMemMandelbrotDataImageID(mandelid))
    SetSpriteImage(1,1)
    Sync()
loop

```

Activity 24.23

You should be able to see a smaller boundary box when the `SetGenerateMipMaps(1)` statement has been added.

25

Drawing

In this Chapter:

- ☐ Using the DrawLine() Statement
- ☐ Drawing Other Basic Shapes
- ☐ Creating a Bezier Curve
- ☐ Creating Wireframe Models from OBJ Files

Drawing Shapes

Introduction

AGK contains only a single statement for drawing directly to the screen. And yet, with that single statement we have enough power to create just about any geometric shape we need.

Drawing a Line

DrawLine()

The `DrawLine()` command will draw a line of a specified colour between two points. The format of this statement is shown in FIG-25.1.

FIG-25.1

`DrawLine()`

`DrawLine ((x1 , y1 , x2 , y2 , r , g , b)`

where

x1, y1 are real numbers giving the coordinates of the starting point of the line.

x2, y2 are real numbers giving the coordinates of the finishing point of the line.

r, g, b are integer values giving the red, green and blue components of the line's colour.

The `DrawLine()` statement has a property similar to the `Print()` statement: it must be executed for each frame since, unlike sprites, text objects and 3D objects, its output is not automatically included in the construction of the next screen frame. The technique required to maintain the display is shown in FIG-25.2.

FIG-25.2

Using `DrawLine()`

```
do
  rem *** Draw a white line from top-left to bottom-right ***
  DrawLine(0,0,100,100,255,255,255)
  Sync()
loop
```

Activity 25.1

Start a new project called *TestDrawLine* and implement and run the code given in FIG-25.2.

Now move the `DrawLine()` statement so that it becomes the first line of code.

How does this affect the program?

Drawing a Dot

An even simpler requirement than a line is drawing a single spot (covering just a few pixels).

We can do this quite easily by using the `DrawLine()` statement with a very small difference between the start and finish points. For example, if we wanted to create a single spot at the position (50,10) we could use the line

```
DrawLine(50,10,50.1,10,255,255,255)
```

or, more correctly, so that the required point is in the middle of the short line:

```
DrawLine(49.95,10,50.05,10,255,255,255)
```

Unfortunately, this code is not sophisticated enough to ensure that the dot covers a single pixel only. The width (in pixels) of the screen on which your app is running (or the width of its window) can be discovered using `GetDeviceWidth()`. When using the percentage system for screen coordinates, a single pixel is equivalent to $(100/\text{GetDeviceWidth}())\%$. With this information, we can create a new function called `DrawDot()` specifically for drawing a single point on the screen:

```
function DrawDot(x as float, y as float,r,g,b)
    pixel# = 100.0/GetDeviceWidth()
    DrawLine(x-pixel#/2,y,x+pixel#/2,y,r,g,b)
endfunction
```

Activity 25.2

Start a new project called *DrawingFunctions*.

Include the function `DrawDot()` (as given above) in the program and write a main section to draw 5000 randomly coloured spots at random positions on the screen.

Test and save your program.

Drawing a Rectangle

To draw a rectangle, we need only specify the coordinates of the top-left ($x1,y1$) and bottom-right ($x2,y2$) corners along with the colour (r,g,b) in which the outline is to be produced. The function has the heading

```
function DrawRectangle(x1 as float, x2 as float, x2 as float,
    y2 as float, r,g,b)
```

followed by code to draw a line from the top-left corner to the top-right corner

```
DrawLine(x1,y1,x2,y1,r,g,b)
```

then from the top-right to the bottom-right:

```
DrawLine(x2,y1,x2,y2,r,g,b)
```

After drawing the remaining two lines we end up with the function as shown in FIG-25.3.

FIG-25.3

Drawing a Rectangle

```
rem *** Draws a rectangle between points(x1,y1) and (x2,y2) ***
rem *** using colour r,g,b ***
function DrawRectangle(x1 as float, y1 as float, x2 as float,
    y2 as float, r,g,b)
    DrawLine(x1,y1,x2,y1,r,g,b)
    DrawLine(x2,y1,x2,y2,r,g,b)
    DrawLine(x2,y2,x1,y2,r,g,b)
    DrawLine(x1,y2,x1,y1,r,g,b)
endfunction
```

Activity 25.3

Add the function *DrawRectangle()* to *DrawingFunctions*.
Test the new function by drawing a yellow rectangle from (10,50) to (80,90) and save your program.

Activity 25.4

Add a new function called *DrawTriangle()* to *DrawingFunctions* which draws a triangle. Test the function by drawing a red triangle with corners at (35,10), (10,30) and (70,60).

Fixing Mandelbrot

In the last chapter we made use of a rectangular outline image to select the zoom-in area of the Mandelbrot set. The problem with this approach was that the rectangle was invisible when small and produced a too-thick outline when large. We can solve this problem by using the *DrawRectangle()* function to show the selected area.

The existing *Mandelbrot* functions responsible for drawing the outline are:

```
CreateBoundary()  
ResizeBoundary()  
DeleteBoundary()
```

These functions load, resize and delete a sprite to create the outline.

We can keep the names and parameters of these routines unchanged but modify their contents so that they now make use of the *DrawRectangle()* function. The new code for *CreateBoundary()* is:

```
rem *** Loads and positions the outline image ***  
function CreateBoundary(x#,y#)  
    global startx#, starty#  
    rem *** Save coords of top-left ***  
    startx# = x#  
    starty# = y#  
    rem *** Draw small rectangle ***  
    DrawRectangle(x#,y#,x#+0.5,y#+0.5,200,200,0)  
endfunction
```

Notice that the function introduces two new global variables, *startx#* and *starty#*. These store the coordinates of the top left corner of the rectangle. The contents of these variables are unchanged as the rectangle is resized.

The *ResizeBoundary()* function (which makes use of the new global variables) is recoded as:

```
rem *** Resize outline ***  
function ResizeBoundary(id)  
    rem *** Calculate image ratio ***  
    w# = GetMemMandelbrotDataImageWidth(id)  
    h# = GetMemMandelbrotDataImageHeight(id)  
    ratio# = w#/h#  
    rem *** Redraw rectangle with ratio of the image ***  
    DrawRectangle(startx#,starty#,GetPointerX(),starty#+  
        (GetPointerX()-startx#)*ratio#,200,200,0)  
endfunction
```

The unusual part of this function is that the y-coordinate of the bottom-right corner is not taken from the pointer device but is calculated to maintain the width to height ratio of the sprite showing the Mandelbrot image.

The final routine, `DeleteBoundary()` is now redundant since there are no sprites or images to delete when the rectangle is no longer visible. However, rather than remove the routine entirely, we will recode it as an empty function:

```
rem *** Delete outline ***
function DeleteBoundary()
endfunction
```

All three of these functions are called by `HandleNewSelection()`. But since we have retained the name, parameters and purpose of the routines in the new coding, there is no need to change any of the lines in `HandleNewSelection()`.

Activity 25.5

Open the projects *Mandelbrot* and *DrawFunctions*.

Copy the code for `DrawRectangle()` from *DrawFunctions* and place it at the end of the project code in *Mandelbrot*. Close the *DrawFunctions* project.

In *Mandelbrot*, modify the code for the functions `CreateBoundary()`, `ResizeBoundary()` and `DeleteBoundary()` to match that given above.

Test and save your project.

One of the aims of good design is that changes to one section of a program should have minimal effect on other sections. The changes made in Activity 25.5 are an excellent example of this. Although we rewrote three functions within our *Mandelbrot* project, no changes were required elsewhere in the code. The primary reason for this being that we did not change function names, parameters, or purpose.

It seems less than efficient to retain a function (`DeleteBoundary()`) which executes no code. However, when the function is removed from the project, we now affect other parts of the program since we must also delete the call to that function at the end of `HandleNewSelection()`.

Activity 25.6

In your *Mandelbrot* project, remove the code for `DeleteBoundary()` and delete the call to `DeleteBoundary()` in `HandleNewSelection()`. Test and save your program.

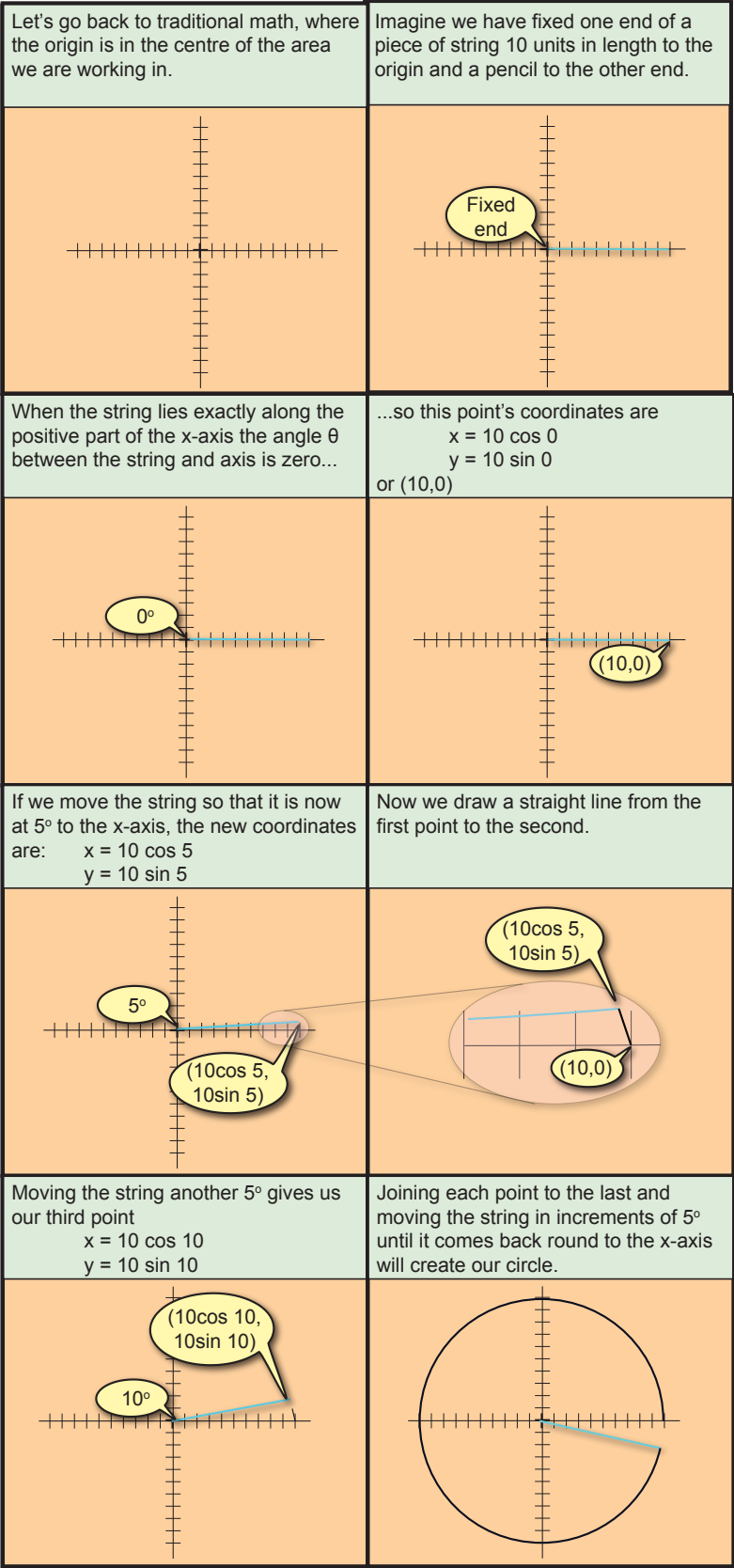
Drawing a Circle

To draw a circle, all you need to remember from your school math is that the points on the circumference of a circle whose centre is at the origin and which has a radius r are defined as:

$$\begin{aligned}x &= r * \cos \theta \\y &= r * \sin \theta\end{aligned}$$

By drawing a line from point to point around the circumference, we can create the outline of a circle. For example, if we wanted to create a circle with a radius of 10 centred on the origin, we could use the techniques described in FIG-25.4.

FIG-25.4
How to Draw a Circle



If we want the circle to be centred elsewhere other than the origin - say at a point (c_x , c_y) - then the formula for points on the circumference becomes:

$$x = r \cos \theta + c_x$$
$$y = r \sin \theta + c_y$$

Any function we create to draw a circle will need the following parameters:

Position of the centre of the circle
Radius
Colour of outline

The function's code is shown in FIG-25.5.

FIG-25.5

DrawCircle() Function

```
rem *** Draws a circle centre (x,y) radius, radius using ***
rem *** colour r,g,b ***
function DrawCircle(x as float,y as float, radius as float, r, g, b)
    rem *** First point on circumference ***
    x# = radius + x
    y# = y
    rem *** Cal remaining points as 8.0 degree steps ***
    for degree# = 8 to 360 step 8.0
        rem *** Store previously calculated point ***
        oldx# = x#
        oldy# = y#
        rem *** Calculate next point on circumference ***
        x# = cos(degree#)*radius + x
        y# = sin(degree#)*radius + y
        rem *** Draw line between previous and new points ***
        drawline(oldx#,oldy#,x#,y#,r,g,b)
    next degree#
endfunction
```

Notice that the points on the circle are 8° apart. The less steps we take to move all the way around the circumference, the quicker the circle is drawn, so 8° is a compromise between accuracy of the circle's outline and the time taken.

Activity 25.7

Add the *DrawCircle()* function given in FIG-25.5 to *DrawingFunctions*.

Modify the main section to draw a red circle at the centre of the screen with a radius of 20.

Test and save your program.

Modify the step size in *DrawCircle()*'s `for` loop from 8 to 1. Does this make the form of the circle appear more accurate?

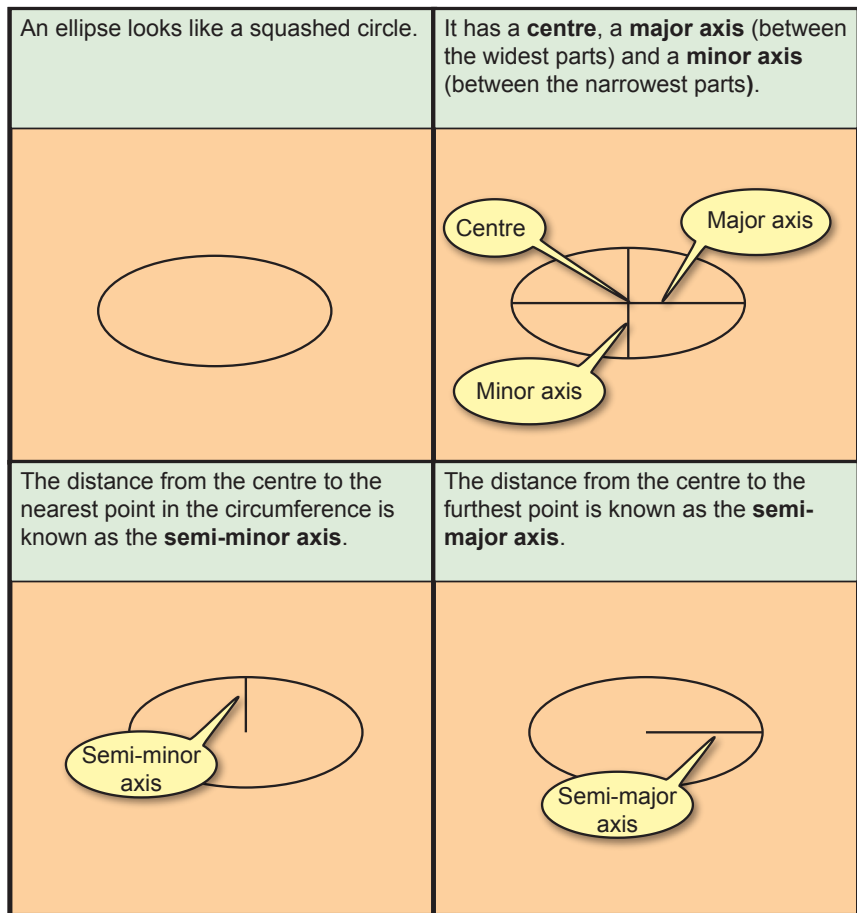
Do NOT save this version of the function.

Drawing an Ellipse

We can think of an ellipse as a circle which has been stretched in one direction. A typical ellipse and its main properties is shown in FIG-25.6.

FIG-25.6

Characteristics of an Ellipse



The simplest way to draw an ellipse is to modify the circle calculation so that the x value is calculated by multiplying $\cos \theta$ by one semi axis length (rather than the radius) and the y value calculated by multiplying $\sin \theta$ by the other semi axis length. This gives us the routine shown in FIG-25.7.

FIG-25.7

The DrawEllipse()
Function

```

rem *** Draws an ellipse centre (x,y) semi-major, ax1, ***
rem *** semi-minor ax2 in colour r,g,b ***
function DrawEllipse(x as float,y as float, ax1 as float,
  ax2 as float, r, g, b)
  rem *** First point on circumference ***
  x# = ax1 + x
  y# = y
  rem *** Cal remaining points as 8.0 degree steps ***
  for degree# = 8 to 360 step 8.0
    rem *** Store previously calculated point ***
    oldx# = x#
    oldy# = y#
    rem *** Calculate next point on the circumference ***
    x# = cos(degree#)*ax1 + x
    y# = sin(degree#)*ax2 + y
    rem *** Draw line between previous and new points ***
    drawline(oldx#,oldy#,x#,y#,r,g,b)
  next degree#
endfunction

```


Activity 25.8

Add the *DrawEllipse()* function given in FIG-25.7 to *DrawingFunctions*.

Change the main section to draw an ellipse with parameter *ax1* set to 30 and *ax2* set to 10. Test and save your program.

Note that it is not possible to draw an ellipse which lies at an angle.

Creating a Data Structure for Basic Shapes

In the last chapter we saw how memblocks can be used to create user-defined data structures. It would be neater and better style if we do the same thing for our basic drawing shapes.

Memblock LineType

The easiest memblock structure to adapt is **LineType** which we created in the previous chapter.

Activity 25.9

Load project *MemblockLine* (Activity 24.12) and add the following function which draws a line based on the details held in a **LineType** variable.

```
rem *** Draws the line on the screen ***
function DrawMemLine(id)
    rem *** Retrieve line details ***
    x1# = GetMemLineStartX(id)
    y1# = GetMemLineStartY(id)
    x2# = GetMemLineFinishX(id)
    y2# = GetMemLineFinishX(id)
    red = GetMemLineColourRed(id)
    green = GetMemLineColourGreen(id)
    blue = GetMemLineColourBlue(id)
    rem *** Draw the line ***
    DrawLine(x1#,y1#,x2#,y2#,red,green,blue)
endfunction
```

Change the main section to read:

```
rem *** Drawing a Memblock Line ***
rem *** Create a LineType structure ***
line1 = CreateMemLineType()
rem *** Assign start, finish and colour to line ***
SetMemLineStart(line1,20.5,34.8)
SetMemLineFinish(line1,76.2,51.6)
SetMemLineColour(line1,255,100,200)
rem *** Draw the line ***
do
    DrawMemLine(line1)
    Sync()
loop
```

Test and save your program.

Activity 25.10

What pre-condition test is missing from the code for `DrawMemLine()`?

Add the pre-condition test to the function, exiting if the condition is not met.

Test and save your project.

Now we are ready to try creating new memblock-based data types for other shapes.

Activity 25.11

Create a new file called *MemUDTLibrary* in your *DrawingFunctions* project and copy the memblock-based functions for `PointType` and `ColourType`. These were created in projects *MemblockPoint* and *MemblockColour*.

Make sure you copy only the functions and not the main sections of each project. Save your project.

The data structure for a circle would be defined conceptually as:

```
type CircleType
  centre as PointType    //Centre of circle
  radius as float        //Radius of circle
  colour as ColourType   //Colour of circumference
endtype
```

The number of bytes required in a memblock for this structure would be:

Field	Type	Bytes
centre	integer (memblock ID)	4
radius	float	4
colour	integer (memblock ID)	4

with the offsets being:

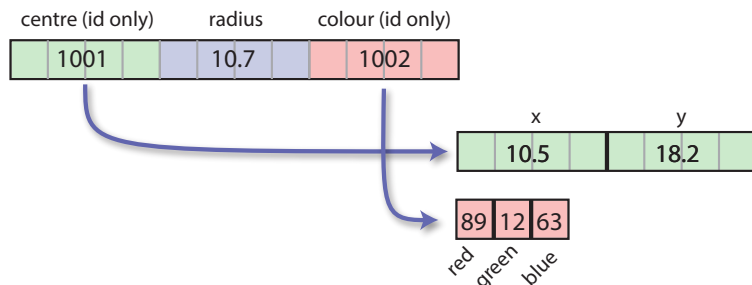
Field	Offset
centre	0
radius	4
colour	8

A visual representation of the structure is shown in FIG-25.8.

FIG-25.8

Representation of a
`CircleType` Structure

Contents of a **CircleType** Item



The basic functions this structure will require are:

<code>CreateMemCircleType()</code>	creates the memblock, initialises the fields and returns the ID assigned to the memblock.
<code>SetMemCircleCentre()</code>	sets the centre of the circle to a specified position.
<code>SetMemCircleRadius()</code>	sets the circle radius.
<code>SetMemCircleColour()</code>	sets the colour to be used when drawing the circle's outline.
<code>DrawMemCircle()</code>	draws the circle on the screen.
<code>GetMemCircleCentreX()</code>	returns the x coordinate of the circle's centre.
<code>GetMemCircleCentreY()</code>	returns the y coordinate of the circle's centre.
<code>GetMemCircleRadius()</code>	returns the radius of the circle.
<code>GetMemCircleColourRed()</code>	returns the red component of the circle's colour.
<code>GetMemCircleColourGreen()</code>	returns the green component of the circle's colour.
<code>GetMemCircleColourBlue()</code>	returns the blue component of the circle's colour.

The code for each of these functions is similar to that in many previous functions and is presented without explanation in FIG-25.9.

FIG-25.9

Implementing CircleType
Using Memblocks

```
rem *****
rem ***          CircleType          ***
rem *****

remstart
Below is the conceptual structure being created in the memblock
type CircleType
    centre    as PointType
    radius    as float
    colour    as ColourType
endtype
remend

rem *** Creates the memblock for CircleType data ***
rem *** Circle created defaults to centre (50,50) ***
rem *** radius 10 and colour red ***
function CreateMemCircleType()
    rem *** Create main memblock ***
    id = CreateMemblock(12)
    rem *** Create memblock for centre and colour fields ***
    centreid = CreateMemPointType(50,50)
    colourid = CreateMemColourType(255,0,0)
    rem *** Store IDs in main memblock ***
```



FIG-25.9

(continued)

Implementing CircleType
Using Memblocks

```

    SetmemblockInt(id,0,centreid)
    SetMemblockInt(id,8,colourid)
    rem *** Store radius ***
    SetMemblockFloat(id,4,10.0)
endfunction id

rem *** Sets the centre of the circle ***
function SetMemCircleCentre(id,x#,y#)
    rem *** IF memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Set the new centre position ***
    centreid = GetMemblockInt(id,0)
    SetMemPoint(centreid,x#,y#)
endfunction

rem *** Sets the circle's radius ***
function SetMemCircleRadius(id, rad#)
    rem *** IF memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** IF rad# < 0,exit function ***
    if rad# < 0
        exitfunction
    endif
    rem *** Set the new radius ***
    SetMemblockFloat(id,4,rad#)
endfunction

rem *** Sets new colour for circle outline ***
function SetMemCircleColour(id,r,g,b)
    rem *** IF memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Set the new colour ***
    colourid = GetMemblockInt(id,8)
    SetMemColour(colourid,r,g,b)
endfunction

rem *** Draws the circle ***
function DrawMemCircle(id)
    rem *** IF memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction
    endif
    rem *** Get drawing colour ***
    r = GetMemCircleColourRed(id)
    g = GetMemCircleColourGreen(id)
    b = GetMemCircleColourBlue(id)
    rem *** Get circle's radius ***
    radius# = GetMemCircleRadius(id)
    rem *** Get centre of circle ***
    centrex# = GetMemCircleCentreX(id)
    centrey# = GetMemCircleCentreY(id)
    rem *** First point on circumference ***
    x# = radius#+centrex#
    y# = centrey#
    rem *** Cal remaining points as 8.0 degree steps ***

```



FIG-25.9

(continued)

Implementing CircleType
Using Memblocks

```

    for degree# = 8 to 360 step 8.0
    rem *** Store previously calculated point ***
    oldx# = x#
    oldy# = y#
    rem *** Calculate next point on circumference ***
    x# = cos(degree#)*radius# + centrex#
    y# = sin(degree#)*radius# + centrey#
    rem *** Draw line between previous and new points ***
    drawline(oldx#,oldy#,x#,y#,r,g,b)
    next degree#
endfunction

rem *** Returns the X coord of circle's centre ***
function GetMemCircleCentreX(id)
    rem *** IF memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Get centre's x coord ***
    centreid = GetMemblockInt(id,0)
    result# = GetMemPointX(centreid)
endfunction result#

rem *** Returns the Y coord of circle's centre ***
function GetMemCircleCentreY(id)
    rem *** IF memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Get centre's y coord ***
    centreid = GetMemblockInt(id,0)
    result# = GetMemPointY(centreid)
endfunction result#

rem *** Returns the circle's radius ***
function GetMemCircleRadius(id)
    rem *** IF memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0.0
    endif
    rem *** Get circle's radius ***
    result# = GetMemblockFloat(id,4)
endfunction result#

rem *** Returns the red component of circle's colour ***
function GetMemCircleColourRed(id)
    rem *** IF memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get circle's red component ***
    colourid = GetMemblockInt(id,8)
    result = GetMemColourRed(colourid)
endfunction result

rem *** Returns the green component of circle's colour ***
function GetMemCircleColourGreen(id)
    rem *** IF memblock doesn't exist, exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif

```



FIG-25.9

(continued)

Implementing `CircleType`
Using Memblocks

```

        rem *** Get circle's green component ***
        colourid = GetMemblockInt(id,8)
        result = GetMemColourGreen(colourid)
    endfunction result

rem *** Returns the blue component of the circle's colour ***
function GetMemCircleColourBlue(id)
    rem *** IF memblock doesn't exist exit ***
    if GetMemblockExists(id) = 0
        exitfunction 0
    endif
    rem *** Get circle's blue component ***
    colourid = GetMemblockInt(id,8)
    result = GetMemColourBlue(colourid)
endfunction result

```

Activity 25.12

Add the code for the *CircleType* functions (as shown in FIG-25.9) to *MemUDTLibrary*.

Change the main section of *DrawingFunctions* to:

```

rem *** Test CircleType ***
#include "MemUDTLibrary.agc"

circle = CreateMemCircleType()
do
    DrawMemCircle(circle)
    Sync()
loop

```

Test and save your program.

Summary

- Use `DrawLine()` to draw a line of a specified colour between two points.
- Use `DrawLine()` in subsequent functions to create other basic shapes.
- You can use memblock structures to define the properties of basic drawing shapes.

Drawing a Simple Bezier Curve

Introduction

As we have seen from the circle and ellipse examples, it is possible to draw a series of straight lines which, when small enough, fool the eye into thinking it is seeing a curved line.

While the curves in a circle and ellipse are somewhat limited, a Bezier curve allows for a greater flexibility in form although its construction is more complex than that required for circles and ellipses. A typical Bezier curve is shown in FIG-25.10.

FIG-25.10

A Bezier Curve



Calculating the Curve

The simplest of the Bezier curves can be defined using three points. These are the start and end points of the line and a control point. It is the position of the control point that determines the line's curvature. The steps in creating a Bezier curve are shown in FIG-25.11.

FIG-25.11

How to Construct a Simple Bezier Curve

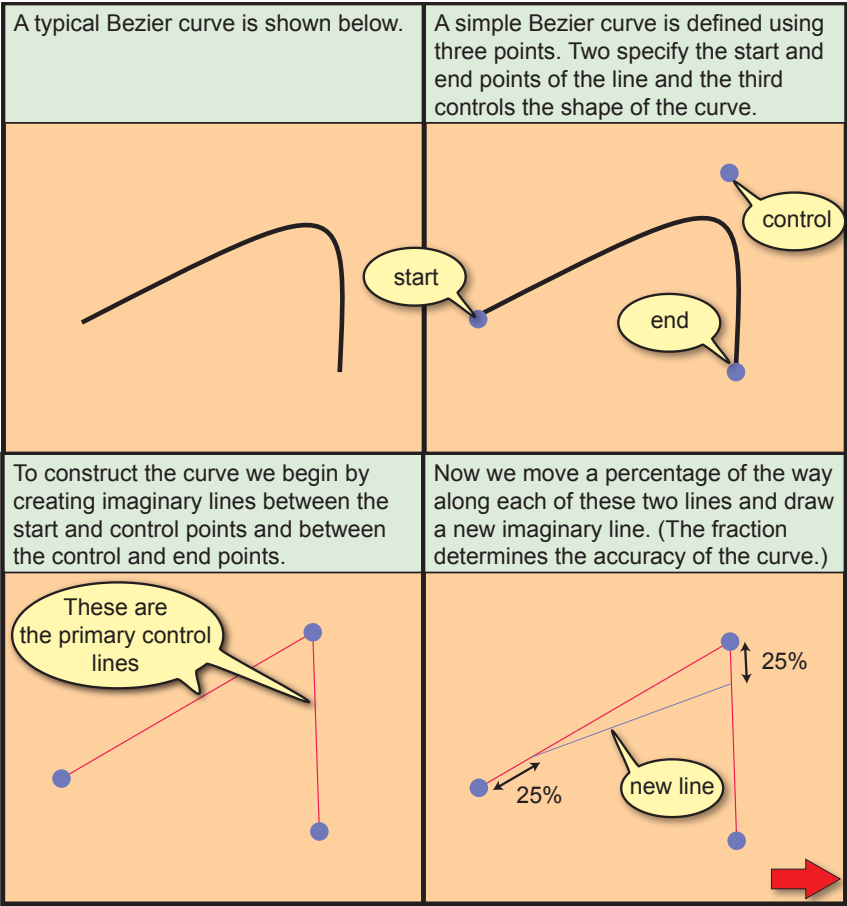
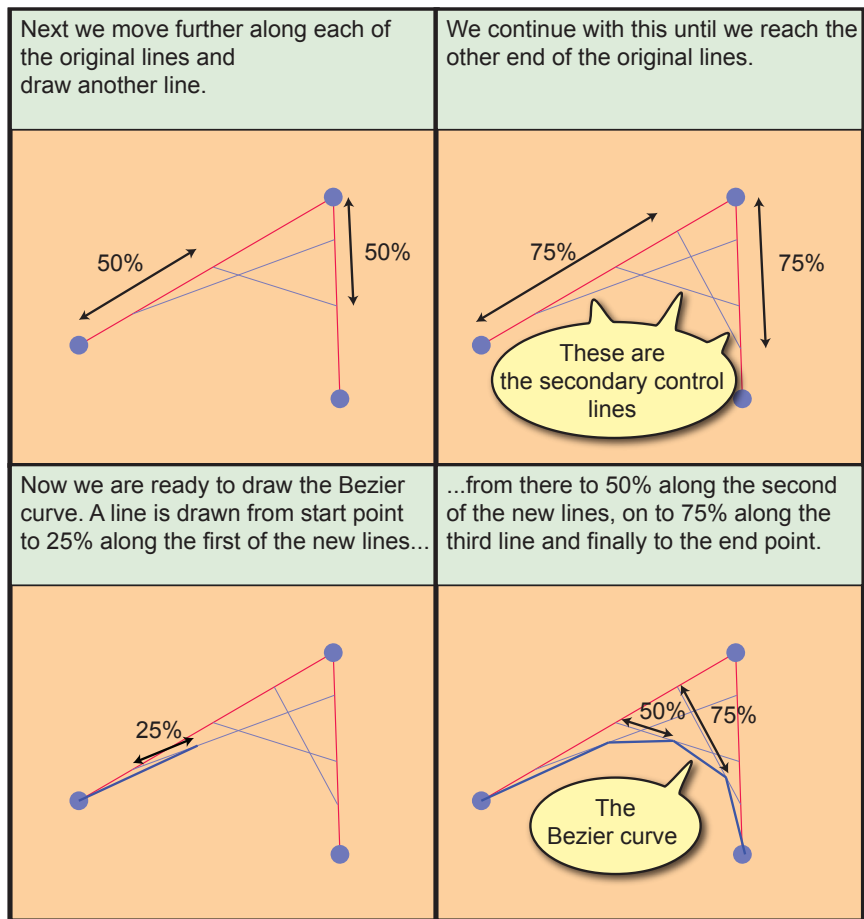


FIG-25.11

(continued)

How to Construct a
Simple Bezier Curve



In the description above we see only three intermediate points along the Bezier curve being calculated. Of course, in reality we need to calculate many more if we are to create a realistic, smooth curve.

To minimise the code required, we will implement the Bezier curve without creating a memblock data structure as we did previously with *CircleType*. The parameters of the *DrawBezierCurve()* function which will draw the curve are:

- The coordinates of the start point
- The coordinates of the finish point
- The coordinates of the control point
- The colour of the line

This list gives us the first line of the function:

```
function DrawBezierCurve(x1 as float, y1 as float, x2 as float,  
    y2 as float, x3 as float, y3 as float, r,g,b)
```

The drawing function requires the following logic:

- Calculate the two primary control lines
- Calculate the step size used on each primary control line
- Calculate and store the set of secondary control lines
- Calculate and draw points on secondary control lines to create the Bezier curve

Before coding the actual logic of the function, it will help if we start by defining

PointType and *LinePointsType* structures:

```
type PointType
  x as float
  y as float
endtype

type LinePointsType
  start as PointType
  fin as PointType
endtype
```

Next we can specify the number of points we are going to calculate on the Bezier curve by defining a constant:

```
#constant POINTS_ON_CURVE = 20
```

The final important item is an array in which to store the start and end points of the secondary control lines:

```
global dim lines[POINTS_ON_CURVE] as LinePointsType
```

Finally, we are ready to implement the logic of the function:

Calculate the two primary control lines

```
rem *** Calculate the two primary control lines ***
c1 as LinePointsType //Control line from start point (x1,y1)
                        ↳ to control point (x3,y3)
c2 as LinePointsType //Control line from control point
                        ↳ (x3,y3) to end point (x2,y2)

c1.start.x = x1
c1.start.y = y1
c1.fin.x = x3
c1.fin.y = y3
c2.start.x = x3
c2.start.y = y3
c2.fin.x = x2
c2.fin.y = y2
```

Calculate the step size used on each primary control line

```
rem *** Calculate step size on each primary control line ***
c1_stepx# = (c1.fin.x - c1.start.x)/POINTS_ON_CURVE
c1_stepy# = (c1.fin.y - c1.start.y)/POINTS_ON_CURVE
c2_stepx# = (c2.fin.x - c2.start.x)/POINTS_ON_CURVE
c2_stepy# = (c2.fin.y - c2.start.y)/POINTS_ON_CURVE
```

Calculate and store the set of secondary control lines

```
rem *** Create and store set of secondary control lines ***
for c = 1 to POINTS_ON_CURVE
  lines[c].start.x = c1.start.x + c*c1_stepx#
  lines[c].start.y = c1.start.y + c*c1_stepy#
  lines[c].fin.x = c2.start.x + c*c2_stepx#
  lines[c].fin.y = c2.start.y + c*c2_stepy#
next c
```

Calculate and draw points on secondary control lines to create the Bezier curve

```
rem *** Calculate and draw points on secondary control lines
↳ to create Bezier curve ***
rem *** Start point for line ***
x# = x1
y# = y1
for c = 1 to POINTS_ON_CURVE
  oldx# = x#
```

```

oldy# = y#
rem *** Retrieve secondary control line & calc step
↳size ***
sc_stepx# = (lines[c].fin.x-lines[c].start.x)/
↳POINTS_ON_CURVE
sc_stepy# = (lines[c].fin.y-lines[c].start.y)/
↳POINTS_ON_CURVE
rem *** Calculate point on retrieved line ***
x# = lines[c].start.x + c*sc_stepx#
y# = lines[c].start.y + c*sc_stepy#
DrawLine(oldx#, oldy#, x#, y#, r,g,b)
next c

```

Activity 25.13

Create a *DrawBezierCurve()* function from the code given above and add it to *DrawingFunctions*, placing the constant, global and type declarations at the start of the project's code.

Try drawing a red Bezier curve using the coordinates (10,50),(80,30),(20,80). The points are listed in the order start, finish and control.

Test and save your program.

Timing Issues

As you can see from the code, creating a Bezier curve takes a considerable amount of calculations and the *DrawBezierCurve()* function needs to be called every time a frame is constructed. Anything we can do to reduce the time taken to draw the curve will have an effect on the overall speed of any app which uses the function. We can start by finding out how long it takes to execute the *DrawBezierCurve()* function 10,000 times by changing the main section of the program to read:

```

CreateText(1,"")
ResetTimer()
for c = 1 to 10000
    DrawBezierCurve(10,50,80,30,20,80,255,0,0)
    //Sync()
next c
SetTextString(1,"Time for 10000 Bezier curves : \"
↳+count$+Str(Timer(),2))
do
    Sync()
loop

```

Notice that the *Sync()* statement after the call to *DrawBezierCurve()* has been disabled. This is because the screen refresh caused by *Sync()* takes much longer than the time to execute the function under test and hides any improvements we may make to the function's execution time. *Sync()* is only called when we want to see the time taken for 10,000 calls to *DrawBezierCurve()*.

Activity 25.14

Modify the main section of code in *DrawingFunctions* to match that given above. Test the program and find out the time taken to execute the *for* loop.

Save your program.

The first modification we could make to the *DrawBezierCurve()* function is to

eliminate the creation of the primary control line variables *c1* and *c2*.

If you examine how these variables are used later in the function's code, you see that they are only needed when accessing the start and end points of the primary control lines. Since the start and end points are actual parameters to the function in the first place, we can easily remove the lines

```
rem *** Calculate the two primary control lines ***
c1 as LineType //Control line from start point (x1,y1) to
               ↳control point(x3,y3)
c2 as LineType //Contol line from control point (x3,y3)to
               ↳end point (x2,y2)

c1.start.x = x1
c1.start.y = y1
c1.fin.x = x3
c1.fin.y = y3
c2.start.x = x3
c2.start.y = y3
c2.fin.x = x2
c2.fin.y = y2
```

and change the lines

```
rem *** Calculate step size on each primary control line ***
c1_stepx# = (c1.fin.x - c1.start.x)/POINTS_ON_CURVE
c1_stepy# = (c1.fin.y - c1.start.y)/POINTS_ON_CURVE
c2_stepx# = (c2.fin.x - c2.start.x)/POINTS_ON_CURVE
c2_stepy# = (c2.fin.y - c2.start.y)/POINTS_ON_CURVE
rem *** Create and store set of secondary control lines ***
for c = 1 to POINTS_ON_CURVE
    lines[c].start.x = c1.start.x + c*c1_stepx#
    lines[c].start.y = c1.start.y + c*c1_stepy#
    lines[c].fin.x = c2.start.x + c*c2_stepx#
    lines[c].fin.y = c2.start.y + c*c2_stepy#
next c
```

to

```
rem *** Calculate step size on each primary control line ***
c1_stepx# = (x3 - x1)/POINTS_ON_CURVE
c1_stepy# = (y3 - y1)/POINTS_ON_CURVE
c2_stepx# = (x2 - x3)/POINTS_ON_CURVE
c2_stepy# = (y2 - y3)/POINTS_ON_CURVE

rem *** Create and store set of secondary control lines ***
for c = 1 to POINTS_ON_CURVE
    lines[c].start.x = x1 + c*c1_stepx#
    lines[c].start.y = y1 + c*c1_stepy#
    lines[c].fin.x = x3 + c*c2_stepx#
    lines[c].fin.y = y3 + c*c2_stepy#
next c
```

This gives us the version of *DrawBezierCurve()* shown in FIG-25.12.

FIG-25.12

The DrawBezierCurve()
Function

```
rem *** Draws a Bezier curve start:(x1,y1) ***
rem *** end: (x2,y2), control:(x3,y3) ***
rem *** colour : r,g,b ***
function DrawBezierCurve(x1 as float, y1 as float, x2 as float,
↳y2 as float, x3 as float, y3 as float, r,g,b)
    rem *** Calculate step size on each primary control line ***
    c1_stepx# = (x3 - x1)/POINTS_ON_CURVE
    c1_stepy# = (y3 - y1)/POINTS_ON_CURVE
```



FIG-25.12

(continued)

The DrawBezierCurve()
Function

```

c2_stepx# = (x2 - x3)/POINTS_ON_CURVE
c2_stepy# = (y2 - y3)/POINTS_ON_CURVE
rem *** Create and store set of secondary control lines ***
for c = 1 to POINTS_ON_CURVE
    lines[c].start.x = x1 + c*c1_stepx#
    lines[c].start.y = y1 + c*c1_stepy#
    lines[c].fin.x = x3 + c*c2_stepx#
    lines[c].fin.y = y3 + c*c2_stepy#
next c
rem *** Draw curve ***
x# = x1
y# = y1
for c = 1 to POINTS_ON_CURVE
    oldx# = x#
    oldy# = y#
    rem *** retrieve secondary control line ***
    sc_stepx# = (lines[c].fin.x-lines[c].start.x)/POINTS_ON_CURVE
    sc_stepy# = (lines[c].fin.y-lines[c].start.y)/POINTS_ON_CURVE
    x# = lines[c].start.x + c*sc_stepx#
    y# = lines[c].start.y + c*sc_stepy#
    DrawLine(oldx#, oldy#, x#, y#, r,g,b)
next c
endfunction

```

Activity 25.15

Change the code for *DrawBezierCurve()* in *DrawingFunctions* to match that given in FIG-25.12.

Test the program. Has the execution time been reduced?

Save your program.

Perhaps surprisingly, the execution time does not appear to change despite the reduction in the code for *DrawBezierCurve()*. However, if we have another look at the function's logic we can see that before the curve can be drawn, a significant amount of calculation needs to be performed. And yet, since the calculations always produce the same results for a given set of parameters, these calculations really only need to be performed once.

So another approach would be to separate the *DrawBezierCurve()*'s code into two separate functions: one to calculate the points on the curve (which only needs to be performed once) and the second to draw a line between these calculated points (which needs to be executed before each call to *Sync()*).

The first routine (which needs a new global array) is shown in FIG-25.13.

FIG-25.13The CalcBezierCurve()
Function

```

global dim Bpoints[50] as PointType

rem *** Calculate points on a Bezier curve. Parameters give ***
rem *** start, end and control points ***

function CalcBezierCurve(x1 as float, y1 as float, x2 as float,
    ↵ y2 as float, x3 as float, y3 as float)
    dim lines[50] as LineType

```



FIG-25.13

(continued)

The CalcBezierCurve()
Function

```

rem *** Calculate step size on each primary control line ***
c1_stepx# = (x3 - x1)/POINTS_ON_CURVE
c1_stepy# = (y3 - y1)/POINTS_ON_CURVE
c2_stepx# = (x2 - x3)/POINTS_ON_CURVE
c2_stepy# = (y2 - y3)/POINTS_ON_CURVE
rem *** Create and store set of secondary control lines ***
for c = 1 to POINTS_ON_CURVE
    lines[c].start.x = x1 + c*c1_stepx#
    lines[c].start.y = y1 + c*c1_stepy#
    lines[c].fin.x = x3 + c*c2_stepx#
    lines[c].fin.y = y3 + c*c2_stepy#
next c
rem *** Calculate points on curve ***
Bpoints[0].x = x1
Bpoints[0].y = y1
for c = 1 to POINTS_ON_CURVE
    rem *** Retrieve secondary control line ***
    sc_stepx# = (lines[c].fin.x-lines[c].start.x)/POINTS_ON_CURVE
    sc_stepy# = (lines[c].fin.y-lines[c].start.y)/POINTS_ON_CURVE
    x# = lines[c].start.x + c*sc_stepx#
    y# = lines[c].start.y + c*sc_stepy#
    Bpoints[c].x = x#
    Bpoints[c].y = y#
next c
endfunction

```

Lines in the new function which match those from the previous version of *DrawBezierCurve()* have been greyed-out in the code.

Notice that a new global array (*Bpoints*) is defined within the function. This array stores the points on the curve and will be used by the second routine to draw the actual curve.

The second new function, used to draw the Bezier curve, is given in FIG-25.14.

FIG-25.14The DrawBezierCurve()
Function (Version 2)

```

rem *** Draw the Bezier curve whose points are stored in ***
rem *** Bpoints, using a line of colour r,g,b ***
function DrawBezierCurve(r,g,b)
    for c = 1 to POINTS_ON_CURVE
        DrawLine(Bpoints[c-1].x,Bpoints[c-1].y, Bpoints[c].x,
            Bpoints[c].y,r,g,b)
    next c
endfunction

```

Activity 25.16

Remove the previous version of *DrawBezierCurve()* from *DrawingFunctions* and add the two new functions given in FIG-25.13 and FIG-25.14.

Modify the code in the main section so that *CalcBezierCurve()* is called immediately before the start of the **for** loop.

Remember to change the parameter list given in the call to *DrawBezierCurve()*.

Test and save your program. Has the execution time been reduced?

Creating a Bezier Curve in Real Time

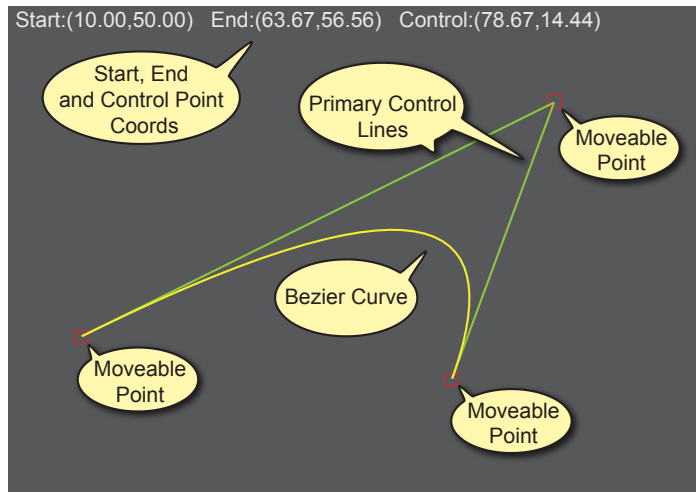
The parameters required to create a specific Bezier curve are not easily calculated by hand. A better approach is to allow the user to edit the curve in real time by enabling him to move the position of the start, end and control points. Once the required curve is achieved, then necessary values can be read off the screen and used in other code.

The Screen Layout

The next program (see screenshot in FIG-25.15) allows the user to drag the three defined points of the curve in real time, with the curve's shape automatically updating as changes are made.

FIG-25.15

Real Time Bezier Curve
Program Screenshot



The moveable points of the curve are indicated using red rectangles. By dragging on these, the curve's start, end and control points can be adjusted. When a red rectangle is selected, the primary lines are displayed in green.

The Program Logic

The overall program logic can be described as:

```
Create the resources required
Set up the initial points of the curve
Calculate the curve
Position adjustment icons (red rectangles)
Display points' coordinates
do
    Check for adjustment to start,end or control points
    IF any point adjusted THEN
        Calculate the curve
        Move adjustment icons and update position displayed
        Update the displayed coordinates
    ENDIF
    Draw the curve
loop
```

The Program Code

This time we will use the top-down approach to create the program. The main section of the program can be written as:

```

rem *** Real Time Bezier Curve ***

rem *** Include drawing definitions and functions ***
#include "DrawingLibrary.agc"

rem *** Points for curve ***
global start as PointType
global fin as PointType
global control as PointType

rem *** Create adjust-icons and text string ***
SetUpResources()
rem *** Set initial points of curve ***
SetUpInitialCurvePoints()
rem *** Calculate curve ***
CalcBezierCurve(start.x,start.y, fin.x,fin.y,
control.x,control.y)
rem *** Position adjustment icons ***
PositionAdjustIcons()
rem *** Display Bezier points' coords ***
SetTextString(1,"Start: (" + Str(start.x,2) + "," + Str(start.y,2) +
␣") End: (" + Str(fin.x,2) + "," + Str(fin.y,2) + ") Control: (" +
␣ + Str(control.x,2) + "," + Str(control.y,2) + ")")
do
  rem *** Check for repositioning of point ***
  moved = CheckPoints()
  if moved <> 0
    rem *** Reposition adjustment icons ***
    PositionAdjustIcons()
    rem *** Display Bezier point coords ***
    SetTextString(1,"Start: (" + Str(start.x,2) + "," +
␣ + Str(start.y,2) + ") End: (" + Str(fin.x,2) + "," + Str(fin.y,2) +
␣ + Str(control.x,2) + "," + Str(control.y,2) + ")")
    rem *** Calculate curve ***
    CalcBezierCurve(start.x,start.y, fin.x,fin.y, control.x,
␣control.y)
  endif
  rem *** Draw curve ***
  DrawBezierCurve(255,255,0)
  Sync()
loop

```

Activity 25.17

Start a new project called *RealTimeBezier* and enter the code given above.

Add a new file called *DrawingLibrary* to the new project.

Open the project *DrawingFunctions* and copy the type, global and function code from that project to *DrawingLibrary*. Make sure you do not copy the main section of the *DrawingFunctions* code.

Close the *DrawingFunctions* project.

Create test stubs for each of the functions called in the code above that are not defined in *DrawingLibrary*.

Test and save your program.

SetUpResources()

This function simply creates the sprite and text resources used by the program and requires the following code:

```
rem *** Creates sprite and text resources ***
function SetUpResources()
    rem *** Create icon for Bezier points ***
    CreateSprite (1,LoadImage("ControlIcon.png"))
    SetSpriteSize(1,3.5,-1)
    CloneSprite(2,1)
    CloneSprite(3,1)
    rem *** Text to display point positions ***
    CreateText(1,"")
    SetTextSize(1,2.5)
endfunction
```

Activity 25.18

Add the above function to *RealTimeBezier* and copy the file *ControlIcon.png* into the *media* folder.

Test and save your program.

SetUpInitialCurvePoints()

The next routine assigns the coordinates of the three Bezier points (start, end and control).

```
rem *** Sets initial value of start,end, and control points ***
function SetUpInitialCurvePoints()
    rem *** Set initial value for start point ***
    start.x = 10
    start.y = 50
    rem *** Set initial value for end point ***
    fin.x = 80
    fin.y = 50
    rem *** Set initial value for control point ***
    control.x = 50
    control.y = 20
endfunction
```

Activity 25.19

Add `SetUpInitialCurvePoints()` to *RealTimeBezier*. Test and save your program.

PositionAdjustIcons()

The red rectangles sprites - referred to as the adjustment icons - need to be positioned at the three Bezier points. The function for this is coded as:

```
rem *** Moves adjustment icons ***
function PositionAdjustIcons()
    rem *** Position sprites ***
    SetSpritePositionByOffset(1,start.x,start.y)
    SetSpritePositionByOffset(2,fin.x,fin.y)
    SetSpritePositionByOffset(3,control.x,control.y)
endfunction
```


Activity 25.20

Add `PositionAdjustIcons()` to *RealTimeBezier*. Test and save your program.

CheckPoints()

The final function of the program is the longest and most complex one. It checks to see if the pointer has been clicked over one of the adjustment icons and if that icon has been moved. It returns a value of 1 if an icon has moved, otherwise zero is returned.

The logic for the routine is:

```
IF pointer not pressed THEN
    Exit function
ENDIF
Get ID of any sprite hit by pointer
IF a sprite has been hit THEN
    Record the position of the sprite's centre
    IF the pointer has just been pressed THEN
        Calculate pointer's offset from sprite's centre
    ENDIF
    Move hit sprite to pointer's position (taking into account initial offset)
    IF the sprite moved THEN
        IF
            start adjust sprite:
                Modify value of start point
            end adjust sprite:
                Modify value of end point
            control adjust sprite:
                Modify value of control sprite
        ENDIF
        Draw the primary control lines
        Set result to 1
    ELSE
        Set result to zero
    ENDIF
    (return value of result)
```

Activity 25.21

Create the function `CheckPoints()` which implements the logic given above and add the function to *RealTimeBezier*.

Test your program by dragging the three adjustment icons and checking that the line is correctly redrawn and the new points' coordinates are displayed.

Are there any problems when moving an adjustment icon?

Save your program.

Fixing the Problem

The trouble with this version of the `CheckPoints()` is that it is very easy to have the pointer slip off an adjustment icon and this causes the reshaping of the curve to stop prematurely.

A better version of the routine would continue moving an icon once it has been

selected until the pointer is released. We can do this by activating icon selection when an icon is hit and deactivating it only when the pointer is released. This requires several changes to our code. First, we need to store the ID of the selected icon in a global variable so that its value will be retained between calls to `CheckPoints()`:

```
rem *** ID of adjust icon being moved ***
rem *** (zero if none ) ***
global id
```

Next we need to make a few changes within the `CheckPoints()` function itself. These are highlighted in the code below:

```
function CheckPoints()
rem *** IF pointer not pressed THEN ***
if GetPointerState()= 0
rem *** Set ID of selected adjust icon to zero ***
id = 0
rem *** Exit function ***
exitfunction
endif
rem *** If no icon selected, check for hit ***
if id = 0
id = GetSpriteHit(GetPointerX(), GetPointerY())
endif
rem *** IF an icon selected THEN ***
if id <> 0
rem *** Record current position of selected sprite ***
oldx# = GetSpriteXByOffset(id)
oldy# = GetSpriteYByOffset(id)
rem *** Calc pointer's offset from sprite's centre ***
if GetPointerPressed() = 1
xoff# = GetPointerX() - GetSpriteXByOffset(id)
yoff# = GetPointerY() - GetSpriteYByOffset(id)
endif
rem *** Move the sprite with pointer taking into ***
rem *** account the pointer's offset ***
newx# = GetPointerX()-xoff#
newy# = GetPointerY()-yoff#
SetSpritePositionByOffset(id,newx#,newy#)
rem *** If the sprite moved, adjust point ***
if newx# <> oldx# or newy# <> oldy#
select id
case 1: // Adjust start point
start.x = newx#
start.y = newy#
endcase
case 2: // Adjust end point
fin.x = newx#
fin.y = newy#
endcase
case 3: //Adjust control point
control.x = newx#
control.y = newy#
endcase
endselect
rem *** Draw primary control lines ***
DrawLine(start.x,start.y,control.x,control.y,100,
200,100)
DrawLine(control.x,control.y,fin.x,fin.y,100,200,100)
endif
endif
endfunction
```

Notice that the function no longer returns a value since the data required by the main

section is stored in the global variable *id*.

Lastly, we need a slight change to the `do..loop` in the main section:

```
do
    rem *** Check for repositioning of point ***
    CheckPoints()
    rem *** IF adjust icon selected THEN ***
    if id <> 0
        rem *** Reposition adjustment icons ***
        PositionAdjustIcons()
        rem *** Display Bezier point coords ***
        SetTextString(1,"Start: (" + Str(start.x,2) + ", "
        ⤵ + Str(start.y,2) + ")   End: (" + Str(fin.x,2) + ", " + Str(fin.y,2) +
        ⤵ + ")   Control: (" + Str(control.x,2) + ", " + Str(control.y,2) +
        ⤵ + ")")
        rem *** Calculate curve ***
        CalcBezierCurve(start.x,start.y, fin.x,fin.y, control.x,
        ⤵ control.y)
    endif
    rem *** Draw curve ***
    DrawBezierCurve(255,255,0)
    Sync()
loop
```

Activity 25.22

Make the necessary changes, as described above, to your *RealTimeBezier* project and check that there is no problem when moving the adjustment icons.

Save your project.

Displaying 3D Models in Wireframe

Introduction

AGK uses OBJ format when saving 3D models to a file. This type of file holds all of the model information in text format. An example of a simple OBJ file is shown in FIG-25.16.

FIG-25.16

The Contents of an OBJ File

```
# Wavefront OBJ exported by MilkShape 3D

v -0.500000 12.000000 7.750000
v -0.500000 0.250000 7.750000
v 15.000000 12.000000 7.750000
v 15.000000 0.250000 7.750000
v 15.000000 12.000000 -7.750000
v 15.000000 0.250000 -7.750000
v -0.500000 12.000000 -7.750000
v -0.500000 0.250000 -7.750000
# 8 vertices

vt 0.000000 1.000000
vt 0.000000 0.000000
vt 1.000000 1.000000
vt 1.000000 0.000000
# 4 texture coordinates

vn 0.000000 0.000000 1.000000
vn 1.000000 0.000000 0.000000
vn 0.000000 0.000000 -1.000000
vn -1.000000 0.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 0.000000 -1.000000 0.000000
# 6 normals

g Box01
s 1
f 1/1/1 2/2/1 3/3/1
f 2/2/1 4/4/1 3/3/1
s 2
f 3/1/2 4/2/2 5/3/2
f 4/2/2 6/4/2 5/3/2
s 1
f 5/1/3 6/2/3 7/3/3
f 6/2/3 8/4/3 7/3/3
s 2
f 7/1/4 8/2/4 1/3/4
f 8/2/4 2/4/4 1/3/4
s 3
f 7/1/5 1/2/5 5/3/5
f 1/2/5 3/4/5 5/3/5
f 2/1/6 8/2/6 4/3/6
f 8/2/6 6/4/6 4/3/6
# 12 triangles in group

# 12 triangles total
```

Each line of text begins with an identifying code of one or two characters.

Lines which start with the symbol # are comments. Other line codes are as follows:

v	vertex
vt	vertex texture UV coordinates
vn	vertex normals
g	group name
s	smoothing group
f	face

This is not an exhaustive description of the OBJ format but it is sufficient to allow us to create a wireframe model of an object.

Two types of entries in an OBJ file are of interest to us. These are the vertex lines, which specify all points on the model and the face lines which tell us how these vertices are joined together. For example, the lines

```
v -0.500000 12.000000 7.750000
v -0.500000 0.250000 7.750000
v 15.000000 12.000000 7.750000
```

tell us that the first three vertices of the model are at points $(-0.5, 12, 7.75)$, $(-0.5, 0.25, 7.75)$ and $(15, 12, 7.75)$. These are numbered as vertices 1, 2 and 3.

Over half way down the file we meet the first face details in the line:

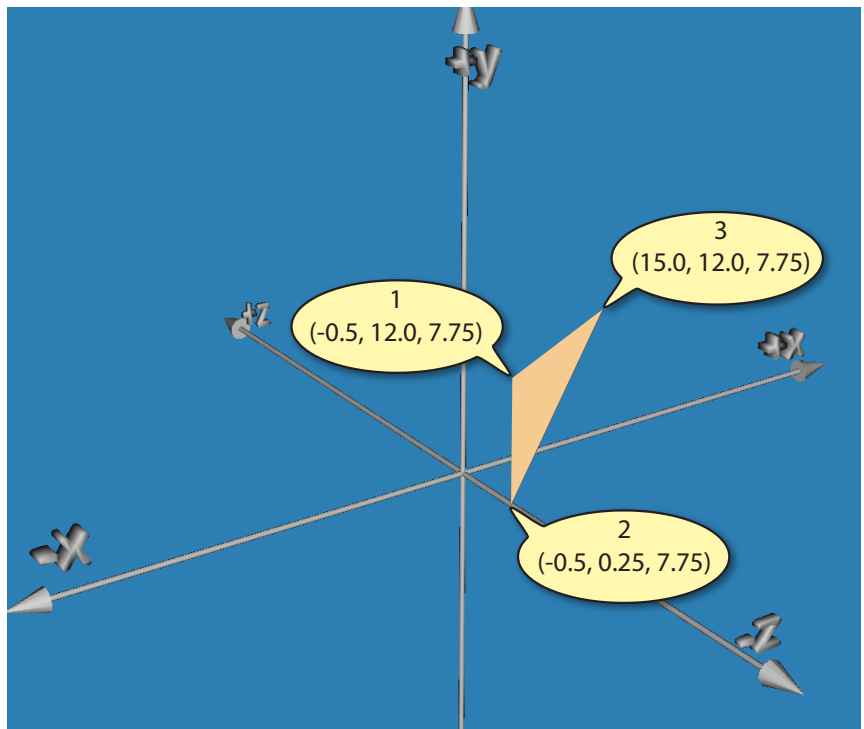
```
f 1/1/1 2/2/1 3/3/1
```

Don't be confused by the three numbers in each group (each number separated by /); only the first number in a group refers to a vertex (the second and third are vertex texture and vertex normal information).

So, the line above says that one face in the model is constructed from joining vertices 1 and 2, 2 and 3, and – since the last vertex must join to the first – 3 and 1 (see FIG-25.17).

FIG-25.17

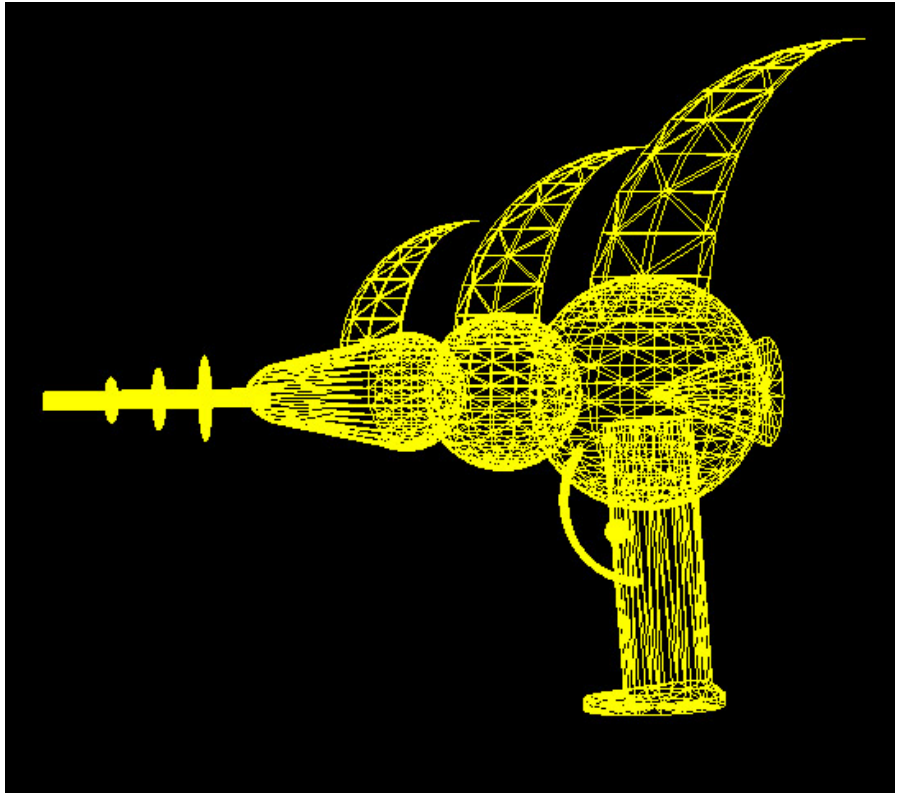
Creating a Face



While a 3D model can be loaded and displayed within an AGK program in a few simple statements, the model is always shown as a solid object. However, by writing a program which reads in the details of an OBJ file and draws lines to represent the edges of every face defined for the model, we can create a wireframe representation (see FIG-25.18).

FIG-25.18

A 3D Model in
Wireframe



Developing the Program Logic

To draw a model in wireframe, we must first select the OBJ file, load all the vertices and faces of the model before drawing lines between the three vertices in each face.

We can summarise this in structured English as:

- 1 Select OBJ file
- 2 Read vertices and faces from file
- 3 Draw lines to construct each face

This short description is worth expanding into more detail using stepwise refinement:

- 1 Select OBJ file
 - 1.1 Create a list of all OBJ files
 - 1.2 Select entry in list
- 2 Read vertices and faces from file
 - 2.1 Open file for reading
 - 2.2 Read line of text from file
 - 2.3 WHILE not EOF DO
 - 2.4 IF lines starts with "v" THEN

```

2.5      Save three vertices
2.6      ELSE IF line starts with "f" THEN
2.7          Save face details
2.8      ENDIF
2.9      Read line of text from file
2.10     ENDWHILE
2.11     Close file

3        Draw lines to construct each face

3.1 FOR each face DO
3.2     Draw lines between vertices 1&2, 2&3, and 3&1 in face
3.3 ENDFOR

```

Implementing the Program

ListOBJFiles()

We begin by implementing the outline logic statement

1.1 Create a list of all OBJ files

as the following function

```

rem *** Returns list of all OBJ files as single string ***
function ListOBJFiles()
    rem *** Create empty list string ***
    list$ = ""
    rem *** Get name of first file in folder ***
    filename$ = GetFirstFile()
    rem *** Get file's extension ***
    fileextension$ = GetStringToken(filename$,".",2)
    rem *** WHILE not looked at all files ***
    while filename$<>""
        rem *** IF OBJ file, add to list ***
        if lower(fileextension$) = "obj"
            list$ = list$+filename$+chr(10)
        endif
        rem *** Look at next file's extension ***
        filename$ = GetNextFile()
        fileextension$ = GetStringToken(filename$,".",2)
    endwhile
endfunction list$

```

Notice that the function inserts a newline character (`chr(10)`) after each OBJ file name. This means that when the string is displayed, each name will be placed on a separate line of the screen.

Activity 25.23

Start a new project called *Wireframe* and enter the code for function `ListOBJFiles()`.

Add a test driver to call the function and display the returned string.

Copy the files *box.obj*, *cylinder.obj*, *sphere.obj*, *cone.obj*, *chair.obj* and *raygun.obj* into the project's *media* folder.

Test your program, ensuring all six files are listed. Save your program.

SelectOBJFile()

The next function implements the logic statement

1.2 Select entry in list

as:

```
rem *** Selects OBJ file to open ***
function SelectOBJFile()
    rem *** Show prompt ***
    CreateText(1,"Click on a file below to open"+Chr(10)+
    ⌨"(no files listed: no OBJs)")
    rem *** List available files ***
    filelist$ = ListOBJFiles()
    CreateText(2,filelist$)
    SetTextPosition(2,20,10)
    rem *** Count number of files ***
    filecount = CountStringTokens(filelist$,Chr(10))
    rem *** Create an invisible sprite over each name ***
    heightofsprite# = GetTextTotalHeight(2)/filecount
    widthofsprite# = GetTextTotalWidth(2)
    for c = 1 to filecount
        CreateSprite(c,0)
        SetSpritePosition(c,20,(c-1)*heightofsprite#+10)
        SetSpriteSize(c,widthofsprite#,heightofsprite#)
        SetSpriteColor(c,200,200,0,100)
    next c
    rem *** Click to select file ***
    selected = 0
    while GetPointerReleased() <> 1 or selected = 0
        if GetPointerPressed() = 1
            selected = GetSpriteHit(GetPointerX(),GetPointerY())
        endif
        Sync()
    endwhile
    rem *** Delete function resources ***
    DeleteAllSprites()
    DeleteAllText()
    rem *** Extract filename ***
    result$ = GetStringToken(filelist$,Chr(10),selected)
endfunction result$
```

This function calls our original function, `ListOBJFiles()` and then displays the returned string. It then places a sprite over each OBJ file listed and waits for the user to click on one of these sprites to select the required file. The selected file name is returned by the function.

Activity 25.24

Add function `SelectOBJFile()` to *Wireframe*.

Change the test driver to call the new function and display the name of the file selected. Test and save your program.

ReadOBJVerticesAndFaces()

With the file required now selected, the next function needs to read the vertices and faces of the model. This means we are implementing the line

2 Read vertices and faces from file

The logic used within the function is described in the previous stepwise refinement of this statement.

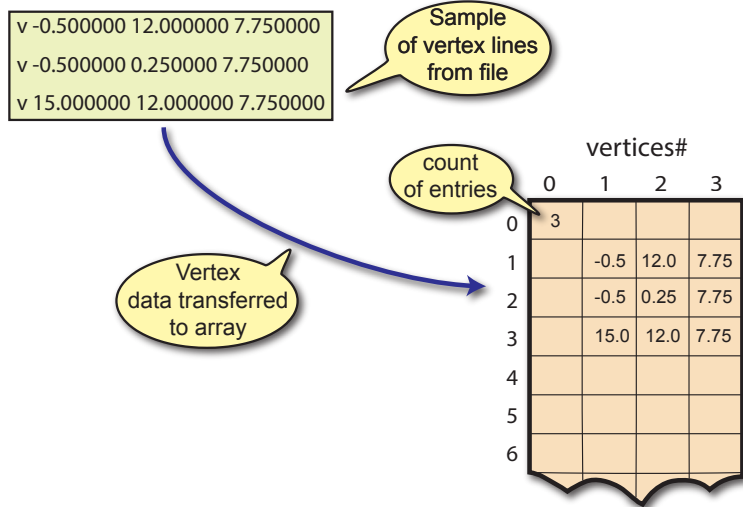
Before we look at the function itself, we need to start by declaring global arrays to hold the vertex and face information:

```
global dim vertices#[6000,3] //Holds vertices (starts at 1)
global dim faces#[6000,3] //Holds faces
```

The *vertices#* array can hold details for up to 6000 vertices (the second dimension gives us space for the x,y and z coordinates of each vertex). The first vertex's details are held in *vertices#[1,1]*, *[1,2]* and *[1,3]* with *vertices#[0,0]* used to contain a count of the number of vertices actually held in the array (see FIG-25.18).

FIG-25.18

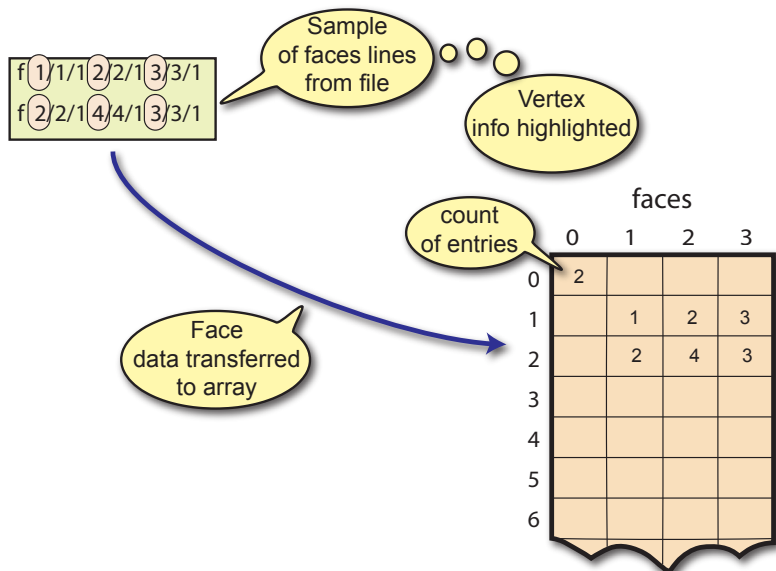
Copying Vertex Details to an Array



The *faces* array holds details for up to 6000 faces. For Milkshape models at least, every face consists of exactly three vertices. The position of a face's vertices in the *vertices#* array are held in the second dimension of the *faces* array. The element *faces#[0,0]* contains a count of the actual number of faces in the model (see FIG-25.19).

FIG-25.19

Copying Face Details to an Array



Now we are ready to create the code for this function:

```
rem *** Read vertices and faces ***
function ReadOBJVerticesAndFaces(file$)
    rem *** Set vertex count to zero ***
    vcount = 0
    rem *** Set face count to zero ***
    fcount = 0
    rem *** Open OBJ file ***
    fileid = OpenToRead(file$)
    rem *** Read first line in file ***
    text$ = ReadLine(fileid)
    rem *** WHILE not EOF, process then read next line ***
    while FileEOF(fileid)=0
        rem *** Get first token in line ***
        linetype$ = GetStringToken(text$," ",1)
        rem *** If it's a vertex, save coords ***
        if linetype$ = "v"
            inc vcount
            for c = 1 to 3
                vertices#[vcount,c]= ValFloat(GetStringToken(text$,
                    ↵ " ",c+1))
            next c
        rem *** If it's a face, save vertex numbers ***
        elseif linetype$ = "f"
            inc fcount
            for c = 1 to 3
                faces[fcount,c] = Val(GetStringToken(text$,
                    ↵ " /",c*3-1))
            next c
        endif
        rem *** Read next line ***
        text$ = ReadLine(fileid)
    endwhile
    rem *** Record vertex count in cell zero ***
    vertices#[0,0] = vcount
    rem *** Record faces count in cell zero ***
    faces[0,0] = fcount
    rem *** Close the file ***
    CloseFile(fileid)
endfunction
```

Activity 25.25

Add the global variables *vertices#* and *faces* as well as the function *ReadOBJVerticesAndFaces()* to *Wireframe*.

Change the test driver to call the new function and display the number of vertices and faces held in the global arrays.

Test and save your program.

DrawWireframe()

Finally, we can make use of the data we have acquired to draw the 3D model. The code for this function is

```
rem *** Draws wireframe of model ***
function DrawWireframe(scale#)
    rem *** FOR each face DO ***
    for face = 1 to faces[0,0]
```

```

rem *** FOR each edge DO ***
for edge = 1 to 3
    rem *** Get starting point ***
    x1# =GetScreenXFrom3D(vertices#[faces[face,edge],1]
    ↵*scale#, vertices#[faces[face,edge],2]*scale#,
    ↵vertices#[faces[face,edge],3]*scale#)
    y1# =GetScreenYFrom3D(vertices#[faces[face,edge],1]
    ↵*scale#,vertices#[faces[face,edge],2]*scale#,
    ↵vertices#[faces[face,edge],3]*scale#)
    rem *** Get end point ***
    fin = edge mod 3 + 1
    x2# = GetScreenXFrom3D(vertices#[faces[face,fin],1]
    ↵*scale#, vertices#[faces[face,fin],2]*scale#,
    ↵vertices#[faces[face,fin],3]*scale#)
    y2# = GetScreenYFrom3D(vertices#[faces[face,fin],1]
    ↵*scale#, vertices#[faces[face,fin],2]*scale#,
    vertices#[faces[face,fin],3]*scale#)
    DrawLine(x1#,y1#,x2#,y2#,200,200,0)
next edge
next face
endfunction

```

The first thing to note about the function is that it takes a scaling factor parameter so that the model can be resized to fit the screen.

At the heart of this function is the `DrawLine()` command which draws lines between the vertices in each face. However, `DrawLine()` operates in only two dimensions, while the model's vertices are defined in three dimensions. To overcome this problem we make use of the `GetScreenXFrom3D()` and `GetScreenYFrom3D()` to convert the model's 3D coordinates to 2D screen coordinates.

Activity 25.26

Add function `DrawWireframe()` to *Wireframe*.

Change the main section of the program to the match the following:

```

rem *** Displaying 3D Models in Wireframe ***

rem *** Global variables ***
global dim vertices#[200,3]      //Holds vertices (starts at 1)
global dim faces[400,3]         //Holds faces

rem *** Get name of OBJ file ***
file$ = SelectOBJFile()
rem *** Read data from file ***
ReadOBJVerticesandFaces(file$)
do
    DrawWireFrame(0.4)
    Sync()
loop

```

Test your program with each of the OBJ files. Save your project.

RotateCamera()

A useful addition to the original design is a new function to rotate the camera about the origin so that the object can be viewed from various angles.

For this we start with a new global variable to hold the camera's angle of rotation about the world y-axis:

```
global angle //Camera angle
```

and the function itself is coded as

```
function RotateCamera()  
    rem *** Calculate camera's new x and z coordinates ***  
    x# = 20*cos(angle)  
    z# = 20*sin(angle)  
    rem *** Reposition camera to match ***  
    SetCameraPosition(1,x#,10,z#)  
    rem *** Make camera point at origin ***  
    SetCameraLookAt(1,0,0,0,0)  
    rem *** Increase angle of rotation ***  
    angle = angle mod 360 + 1  
endfunction
```

Activity 25.27

Add the global variable *angle* and the function `RotateCamera()` to *Wireframe*.

In the `do..loop` of the main section, add a call to `RotateCamera()`.

Test and save your program.

Activity 25.28

You can see from FIG-25.18 and 25.19 that a whole column in both the *vertices#* and *faces* arrays is unused. When the array is large, this represents a significant waste of memory space.

Modify your program so that no column remains unused in either array.

Test and save your program.

Solutions

Activity 25.1

When the `DrawLine()` statement is moved out of the loop, the line is no longer visible. It is draw only in the first frame and since the frame changes so quickly, you are unlikely to see it even for a brief moment.

Activity 25.2

Code for *DrawingFunctions*:

```
rem *** Test dot drawing ***

rem *** Draw 5000 random dots ***
for c = 1 to 5000
    DrawDot(Random(0,100),Random(0,100),Random(0,255),
        Random(0,255), Random(0,255))
next c
Sync()
do
loop

rem *** Draws a single dot at (x,y) ***
rem *** using colour r,g,b ***
function DrawDot(x as float, y as float,r,g,b)
    pixel# = 100.0/GetDeviceWidth()
    DrawLine(x-pixel#/2,y,x+pixel#/2,y,r,g,b)
endfunction
```

Activity 25.3

Updated code for *DrawingFunctions*:

```
rem *** Test rectangle drawing ***
do
    DrawRectangle(10,50,80,90,255,255,0)
    Sync()
loop

rem *** Draws a single dot at (x,y) ***
rem *** using colour r,g,b ***
function DrawDot(x as float, y as float,r,g,b)
    pixel# = 100.0/GetDeviceWidth()
    DrawLine(x-pixel#/2,y,x+pixel#/2,y,r,g,b)
endfunction

rem *** Draws a rectangle between points(x1,y1) ***
rem *** and (x2,y2) using colour r,g,b ***
function DrawRectangle(x1 as float, y1 as float,
    x2 as float,y2 as float, r,g,b)
    DrawLine(x1,y1,x2,y1,r,g,b)
    DrawLine(x2,y1,x2,y2,r,g,b)
    DrawLine(x2,y2,x1,y2,r,g,b)
    DrawLine(x1,y2,x1,y1,r,g,b)
endfunction
```

Activity 25.4

New code for *DrawFunctions*:

(NOTE: existing functions have been omitted from the listing)

```
rem *** Test triangle drawing ***
do
    DrawTriangle(35,10,10,30,70,60,255,0,0)
    Sync()
loop

rem *** Draws a triangle between points (x1.y1), ***
rem *** (x2,y2), (x3,y3) using colour r,g,b ***
function DrawTriangle(x1 as float, y1 as float, x2 as
    float, y2 as float, x3 as float, y3 as float, r,g,b)
    DrawLine(x1,y1,x2,y2,r,g,b)
    DrawLine(x2,y2,x3,y3,r,g,b)
    DrawLine(x3,y3,x1,y1,r,g,b)
endfunction
```

Activity 25.5

No code solution required.

You should notice that the outline of the selected area is visible even when a small area is selected and that the lines remain the same thickness as the selected area changes size.

Activity 25.6

No code solution required.

The program should operate exactly as before.

Activity 25.7

New code for *DrawFunctions*:

```
rem *** Test circle drawing ***
do
    DrawCircle(50,50,20,255,0,0)
    Sync()
loop

rem *** Draws a circle centre (x,y) radius, radius
using ***
rem *** colour r,g,b ***
function DrawCircle(x as float,y as float,
    radius as float, r, g, b)
    rem *** First point on circumference ***
    x# = radius + x
    y# = y
    rem *** Calc remaining points as 8.0 deg steps ***
    for degree# = 8 to 360 step 8.0
        rem *** Store previously calculated point ***
        oldx# = x#
        oldy# = y#
        rem *** Calc next point on circumference ***
        x# = cos(degree#)*radius + x
        y# = sin(degree#)*radius + y
        rem *** Draw line from old to new point ***
        DrawLine(oldx#,oldy#,x#,y#,r,g,b)
    next degree#
endfunction
```

To change to 1 degree steps, change the line

```
for degree# = 8 to 360 step 8.0
```

to

```
for degree# = 1 to 360
```

There should be little or no change to the smoothness of the circle's outline.

Activity 25.8

New code for *DrawFunctions*:

```
rem *** Test ellipse drawing ***
do
    DrawEllipse(50,50,30,10,255,255,255)
    Sync()
loop

rem *** Draws an ellipse centre (x,y) semi-major,
ax1, ***
rem *** semi-minor ax2 in colour r,g,b ***
function DrawEllipse(x as float,y as float, ax1 as
    float,ax2 as float, r, g, b)
    rem *** First point on circumference ***
    x# = ax1 + x
    y# = y
    rem *** Calc remaining points as 8.0 deg steps ***
    for degree# = 8 to 360 step 8.0
        rem *** Store previously calculated point ***
        oldx# = x#
        oldy# = y#
        rem *** Calc next point on circumference ***
        x# = cos(degree#)*ax1 + x
```

```

y# = sin(degree#)*ax2 + y
rem *** Draw line from old to new point ***
DrawLine(olddx#,oldy#,x#,y#,r,g,b)
next degree#
endfunction

```

Activity 25.9

No solution required

Activity 25.10

As with all other functions which accept a memblock ID, we must check that the memblock exists. This means that the function `DrawMemLine()` should begin with the code:

```

function DrawMemLine(id)
rem *** If memblock does not exist, exit ***
if GetMemblockExists(id) = 0
exitfunction
endif

```

Activity 25.11

Make sure `DrawingFunctions` is the active project.

Add a new file to the project. There are three ways to do this:

Press `Ctrl+Shift+N`

Press the new file button in the task bar.

then select Empty File.

Select `File|New|Empty file` from the menu bar.



Enter the name of the file as *MemUDTLibrary*.

Open project *MemblockPoint* and copy the functions listed.

Paste the code into *MemUDTLibrary*.

Close project *MemblockPoint*.

Open project *MemblockColour* and copy the functions listed.

Paste the code into *MemUDTLibrary*.

Close project *MemblockColour*.

Select `File|Save everything`

Activity 25.12

No solution required.

Activity 25.13

The updated version of *DrawFunctions*:

```

rem *** Test Bezier curve ***

rem *** Number of points calculated for curve ***
#constant POINTS_ON_CURVE = 20

rem *** Types used by Bezier curve ***
type PointType
x as float
y as float
endtype

type LinePointsType
start as PointType
fin as PointType
endtype

do
DrawBezierCurve(10,50,80,30,20,80,255,0,0)
Sync()
loop

rem *** Draws a single dot at (x,y) ***
rem *** using colour r,g,b ***
function DrawDot(x as float, y as float,r,g,b)
pixel# = 100.0/GetDeviceWidth()
DrawLine(x-pixel#/2,y,x+pixel#/2,y,r,g,b)

```

```

endfunction

rem *** Draws a rectangle between points (x1,y1) and
(x2,y2) using colour r,g,b ***
function DrawRectangle(x1 as float, y1 as float,
x2 as float,y2 as float, r,g,b)
DrawLine(x1,y1,x2,y1,r,g,b)
DrawLine(x2,y1,x2,y2,r,g,b)
DrawLine(x2,y2,x1,y2,r,g,b)
DrawLine(x1,y2,x1,y1,r,g,b)
endfunction

```

```

rem *** Draws a triangle between points (x1,y1), ***
rem *** (x2,y2), (x3,y3) using colour r,g,b ***
function DrawTriangle(x1 as float, y1 as float, x2 as
float, y2 as float, x3 as float, y3 as float, r,g,b)
DrawLine(x1,y1,x2,y2,r,g,b)
DrawLine(x2,y2,x3,y3,r,g,b)
DrawLine(x3,y3,x1,y1,r,g,b)
endfunction

```

```

rem *** Draws circle, centre: (x,y) radius: radius
using colour r,g,b ***
function DrawCircle(x as float,y as float,
radius as float, r, g, b)
rem *** First point on circumference ***
x# = radius + x
y# = y
rem *** Calc remaining points as 8.0 deg steps ***
for degree# = 8.0 to 360.0 step 8.0
rem *** Store previously calculated point ***
olddx# = x#
oldy# = y#
rem *** Calc next point on circumference ***
x# = cos(degree#)*radius + x
y# = sin(degree#)*radius + y
rem *** Draw line from old to new point ***
DrawLine(olddx#,oldy#,x#,y#,r,g,b)
next degree#
endfunction

```

```

rem *** Draws an ellipse centre (x,y) semi-major,
ax1, ***
rem *** semi-minor ax2 in colour r,g,b
***
function DrawEllipse(x as float,y as float, ax1 as
float,ax2 as float, r, g, b)
rem *** First point on circumference ***
x# = ax1 + x
y# = y
rem *** Calc remaining points as 8.0 deg steps ***
for degree# = 8 to 360 step 8.0
rem *** Store previously calculated point ***
olddx# = x#
oldy# = y#
rem *** Calc next point on circumference ***
x# = cos(degree#)*ax1 + x
y# = sin(degree#)*ax2 + y
rem *** Draw line from old to new point ***
DrawLine(olddx#,oldy#,x#,y#,r,g,b)
next degree#
endfunction

```

```

rem *** Draws a Bezier curve start:(x1,y1) ***
rem *** end: (x2,y2), control:(x3,y3) ***
rem *** colour : r,g,b ***
function DrawBezierCurve(x1 as float, y1 as float,
x2 as float, y2 as float, x3 as float, y3 as float,
r,g,b)
rem *** Secondary control lines ***
dim lines[POINTS_ON_CURVE] as LinePointsType
rem *** Calculate two primary control lines ***
c1 as LinePointsType //Control line from start
point (x1,y1) to control point(x3,y3)
c2 as LinePointsType //Contol line from control
point (x3,y3) to end point (x2,y2)
c1.start.x = x1
c1.start.y = y1
c1.fin.x = x3
c1.fin.y = y3
c2.start.x = x3
c2.start.y = y3
c2.fin.x = x2
c2.fin.y = y2
rem *** Calculate step size on each primary
control line ***
c1_stepx# = (c1.fin.x - c1.start.x)/POINTS_ON_CURVE
c1_stepy# = (c1.fin.y - c1.start.y)/POINTS_ON_CURVE
c2_stepx# = (c2.fin.x - c2.start.x)/POINTS_ON_CURVE
c2_stepy# = (c2.fin.y - c2.start.y)/POINTS_ON_CURVE
rem *** Create and store set of secondary control

```

```

lines ***
for c = 1 to POINTS_ON_CURVE
    lines[c].start.x = c1.start.x + c*c1_stepx#
    lines[c].start.y = c1.start.y + c*c1_stepy#
    lines[c].fin.x = c2.start.x + c*c2_stepx#
    lines[c].fin.y = c2.start.y + c*c2_stepy#
next c
rem *** Calculate and draw points on secondary
    ⚡control lines to create Bezier curve ***
rem *** Start point for line ***
x# = x1
y# = y1
for c = 1 to POINTS_ON_CURVE
    oldx# = x#
    oldy# = y#
    rem *** Retrieve secondary control line & calc
        ⚡step size ***
    sc_stepx# = (lines[c].fin.x-lines[c].start.x)/
        ⚡POINTS_ON_CURVE
    sc_stepy# = (lines[c].fin.y-lines[c].start.y)/
        ⚡POINTS_ON_CURVE
    rem *** Calculate point on retrieved line ***
    x# = lines[c].start.x + c*sc_stepx#
    y# = lines[c].start.y + c*sc_stepy#
    DrawLine(oldx#, oldy#, x#, y#, r,g,b)
next c
endfunction

```

Activity 25.14

Obviously, the exact time taken will depend on the hardware being used but a typical time might be 6.25 seconds.

Activity 25.15

There is no (or very little) change in the execution time.

Activity 25.16

The main section of *DrawFunctions* should now be:

```

CreateText(1,"")
ResetTimer()
CalcBezierCurve(10,50,80,30,20,80)
for c = 1 to 10000
    DrawBezierCurve(255,0,0)
    //Sync()
next c
SetTextString(1,"Time for 10000 Bezier curves : \"
⚡+count$+Str(Timer(),2))
do
    Sync()
loop

```

Even with the calculations removed to a separate function, the execution time is reduced by about only 0.1 seconds.

Activity 25.17

Contents of *DrawingLibrary* (with added comments):

```

rem *****
rem *** Functions using DrawLine() ***
rem *****

rem ***** Constants *****
rem *** Number of points calculated for Bezier curve ***
#constant POINTS_ON_CURVE = 20

rem *****Types*****
type PointType
    x as float
    y as float
endtype

type LineType
    start as PointType
    fin as PointType
endtype

rem *****Globals*****
rem *** Contains points on the Bezier curve ***
global dim Bpoints[POINTS_ON_CURVE] as PointType

```

```

rem *****Functions*****

rem *** Draws a rectangle. top-left:(x1,y1) ***
rem *** bottom-right:(x2,y2), colour: r,g,b ***
function DrawRectangle(x1 as float, y1 as float, x2 as float, y2 as float, r, g, b)
    DrawLine(x1,y1,x2,y1,r,g,b)
    DrawLine(x2,y1,x2,y2,r,g,b)
    DrawLine(x2,y2,x1,y2,r,g,b)
    DrawLine(x1,y2,x1,y1,r,g,b)
endfunction

rem *** Draws a circle. centre:(x,y) ***
rem *** radius:radius, colour:(r,g,b) ***
function DrawCircle(x as float, y as float, radius as float, r, g, b)
    rem *** First point on circumference ***
    x# = radius + x
    y# = y
    rem *** Calc remaining points in 8 deg steps ***
    for degree# = 8 to 360 step 8.0
        oldx# = x#
        oldy# = y#
        x# = cos(degree#)*radius + x
        y# = sin(degree#)*radius + y
        DrawLine(oldx#,oldy#,x#,y#,r,g,b)
    next degree#
endfunction

rem *** Draws an ellipse centre:(x1,y1) ***
rem *** x-semiaxis:ax1, y-semiaxis:ax2 ***
rem *** colour:(r,g,b) ***
function DrawEllipse(x as float, y as float, ax1 as float, ax2 as float, r, g, b)
    rem *** First point on circumference ***
    x# = ax1 + x
    y# = y
    rem *** Calc remaining points as 8deg steps ***
    for degree# = 8 to 360 step 8.0
        oldx# = x#
        oldy# = y#
        x# = cos(degree#)*ax1 + x
        y# = sin(degree#)*ax2 + y
        DrawLine(oldx#,oldy#,x#,y#,r,g,b)
    next degree#
endfunction

rem *** Calculates the points on a Bezier curve ***
rem *** start(x1,y1),end:(x2,y2),control:(x3,y3)***
function CalcBezierCurve(x1 as float, y1 as float,
    ⚡x2 as float, y2 as float, x3 as float, y3 as float)
    dim lines[POINTS_ON_CURVE] as LineType
    rem *** Calc step size on each primary control
        ⚡line ***
    c1_stepx# = (x3 - x1)/POINTS_ON_CURVE
    c1_stepy# = (y3 - y1)/POINTS_ON_CURVE
    c2_stepx# = (x2 - x3)/POINTS_ON_CURVE
    c2_stepy# = (y2 - y3)/POINTS_ON_CURVE
    rem *** Create secondary control lines ***
    for c = 1 to POINTS_ON_CURVE
        lines[c].start.x = x1 + c*c1_stepx#
        lines[c].start.y = y1 + c*c1_stepy#
        lines[c].fin.x = x3 + c*c2_stepx#
        lines[c].fin.y = y3 + c*c2_stepy#
    next c
    rem *** Calculate curve points ***
    Bpoints[0].x = x1
    Bpoints[0].y = y1
    for c = 1 to POINTS_ON_CURVE
        rem *** retrieve secondary control line ***
        sc_stepx# = (lines[c].fin.x - lines[c].start.x)/
            ⚡POINTS_ON_CURVE
        sc_stepy# = (lines[c].fin.y - lines[c].start.y)/
            ⚡POINTS_ON_CURVE
        x# = lines[c].start.x + c*sc_stepx#
        y# = lines[c].start.y + c*sc_stepy#
        Bpoints[c].x = x#
        Bpoints[c].y = y#
    next c
endfunction

rem *** Draws Bezier curve using colour (r,g,b) ***
rem *** Points in global variable Bpoints[] ***
function DrawBezierCurve(r,g,b)
    for c = 1 to POINTS_ON_CURVE
        DrawLine(Bpoints[c-1].x,Bpoints[c-1].y,
            ⚡Bpoints[c].x,Bpoints[c].y,r,g,b)
    next c
endfunction

```

Code for *RealTimeBezier*:

```
rem *** Real Time Bezier Curve ***

rem *** Include drawing definitions and functions ***
#include "DrawingLibrary.agc"

rem *** Points for curve ***
global start as PointType
global fin as PointType
global control as PointType

rem *** Create adjust-icons and text string ***
SetUpResources()
rem *** Set initial points of curve ***
SetUpInitialCurvePoints()
rem *** Calculate curve ***
CalcBezierCurve(start.x,start.y, fin.x,fin.y,
control.x,control.y)
rem *** Position adjustment icons ***
PositionAdjustIcons()
rem *** Display Bezier points' coords ***
SetTextString(1,"Start: ("&Str(start.x,2)&"+","&
Str(start.y,2)&") End: ("&Str(fin.x,2)&"+","&Str(fin.y,2)&")
Control: ("&Str(control.x,2)&"+","&Str(control.y,2)&")")
do
rem *** Check for repositioning of point ***
moved = CheckPoints()
if moved <> 0
rem *** Reposition adjustment icons ***
PositionAdjustIcons()
rem *** Display Bezier point coords ***
SetTextString(1,"Start: ("&Str(start.x,2)&"+","&
Str(start.y,2)&") End: ("&Str(fin.x,2)&"+","&
Str(fin.y,2)&") Control: ("&Str(control.x,2)&"+","&
Str(control.y,2)&")")
rem *** Calculate curve ***
CalcBezierCurve(start.x,start.y, fin.x,fin.y,
control.x, control.y)
endif
rem *** Draw curve ***
DrawBezierCurve(255,255,0)
Sync()
loop

function SetUpResources()
endfunction

function SetUpInitialCurvePoints()
endfunction

function PositionAdjustIcons()
endfunction

function CheckPoints()
endfunction
```

The program will not run because the text resource referred to in the main program has not yet been created.

Activity 25.18

There is a problem with the current version of AGK (108 beta11) such that the global declaration of the array *Bpoints[]* is not recognised when the program is run.

If you have this problem, cut the line in which *Bpoints[]* is declared and paste it as the first line within the function *CalcBezierCurve()*. This will solve the problem.

When the program is run you should see a display of the start, end and control points of the curve (these are all zero). Also, in the top-left corner are the adjustment icons (all three are in the same position).

Activity 25.19

The curve is now displayed in yellow, but the adjust icons are still in the top-left corner.

Activity 25.20

The adjustment icons are now correctly position at the start, end and control points on the line.

Activity 25.21

The code for *CheckPoints()* is:

```
function CheckPoints()
rem *** IF pointer not pressed, exit ***
if GetPointerState() = 0
exitfunction 0
endif
rem *** Get ID of any sprite hit ***
id = GetSpriteHit(GetPointerX(),GetPointerY())
rem *** IF a sprite has been hit THEN ***
if id <> 0
rem *** Record position of sprite's centre ***
oldx# = GetSpriteXByOffset(id)
oldy# = GetSpriteYByOffset(id)
rem *** IF pointer just pressed THEN ***
if GetPointerPressed() = 1
rem *** Calc pointer offset from sprite
centre ***
xoff# = GetPointerX() -
GetSpriteXByOffset(id)
yoff# = GetPointerY() -
GetSpriteYByOffset(id)
endif
rem *** Move hit sprite to pointer position ***
newx# = GetPointerX()-xoff#
newy# = GetPointerY()-yoff#
SetSpritePositionByOffset(id,newx#,newy#)
rem *** IF the sprite moved THEN
if newx# <> oldx# or newy# <> oldy#
select id
case 1://start adjust sprite:
start.x = newx#
start.y = newy#
endcase
case 2: // Adjust end point
fin.x = newx#
fin.y = newy#
endcase
case 3: //Adjust control point
control.x = newx#
control.y = newy#
endcase
endselect
endif
rem *** Draw primary control lines ***
DrawLine(start.x,start.y,control.x,control.y,
100,200,100)
DrawLine(control.x,control.y,fin.x,fin.y,
100,200,100)
rem *** Set result to 1 ***
result = 1
else
rem *** Set result to 0 ***
result = 0
endif
endfunction result
```

When the program is run, the pointer has to be moved quite slowly otherwise it moves off the adjustment icon and redrawing of the line is terminated.

Activity 25.22

With the changes in place, the selected icon will continue to move until the pointer is released.

Activity 25.23

The code for *WireFrame* (function and driver):

```
rem *** Displaying 3D Models in Wireframe ***
rem *** ListOBJFiles test driver ***
do
Print(ListOBJFiles())
Sync()
loop
```



```

rem *** Returns list of all OBJ files as single
↳string ***
function ListOBJFiles()
    rem *** Create empty list string ***
    list$ = ""
    rem *** Get name of first file in folder ***
    filename$ = GetFirstFile()
    rem *** Get file's extension ***
    fileextension$ = GetStringToken(filename$,".",2)
    rem *** WHILE not looked at all files ***
    while filename$<>" "
        rem *** IF OBJ file, add to list ***
        if lower(fileextension$) = "obj"
            list$ = list$+filename$+chr(10)
        endif
        rem *** Look at next file's extension ***
        filename$ = GetNextFile()
        fileextension$ = GetStringToken(filename$,".",2)
    endwhile
endfunction list$

```

Activity 25.24

Wireframe test driver for `SelectOBJFile()`:

```

file$ = SelectOBJFile()
do
    Print(file$)
    Sync()
loop

```

Each file listed is shown in a light orange area (which appear to merge into a single box). These are the sprites used to help select an entry from the list.

Activity 25.25

New *Wireframe* test driver:

```

rem *** Global variables ***
global dim vertices#[6000,3] //Holds vertices
                                ↳(starts at 1)
global dim faces[6000,3] //Holds faces

file$ = SelectOBJFile()
ReadOBJVerticesAndFaces(file$)
do
    Print(file$)
    Print("Vertices: "+Str(vertices#[0,0],0)+
    ↳ " Faces: "+Str(faces[0,0]))
    Sync()
loop

```

Activity 25.26

The OBJ models should now appear in wireframe mode.

You may wish to adjust the scaling factor for some models.

Activity 25.27

The final code for *Wireframe*'s main section:

```

rem *** Displaying 3D Models in Wireframe ***

rem *** Global variables ***
global dim vertices#[6000,3] //Holds vertices (starts
at 1)
global dim faces[6000,3] //Holds faces
global angle //Camera angle

rem *** Get name of OBJ file ***
file$ = SelectOBJFile()
rem *** Read data from file ***
ReadOBJVerticesAndFaces(file$)
do
    DrawWireFrame(0.4)
    RotateCamera()
    Sync()
loop

```

The camera now rotates to allow you to see each model from a varying angle. Some models are offset from the origin and therefore change position on the screen as the camera rotates.

Activity 25.28

Obviously, we must start by reducing the number of columns in the arrays:

```

global dim vertices#[6000,2] //Holds vertices ***
global dim faces[6000,2] //Holds faces ***

```

Now we must ensure that when the vertex and face details are stored that the second subscript is between 0 and 2 rather than 1 and 3.

The required changes to `ReadOBJVerticesAndFaces()` are highlighted:

```

rem *** Read vertices and faces ***
function ReadOBJVerticesAndFaces(file$)
    rem *** Set vertex count to zero ***
    vcount = 0
    rem *** Set face count to zero ***
    fcount = 0
    rem *** Open OBJ file ***
    fileid = OpenToRead(file$)
    rem *** Read first line in file ***
    text$ = ReadLine(fileid)
    rem *** WHILE not EOF, process then read next line
    ***
    while FileEOF(fileid)=0
        rem *** Get first token in line ***
        linetype$ = GetStringToken(text$," ",1)
        rem *** If it's a vertex, save coords ***
        if linetype$ = "v"
            inc vcount
            for c = 1 to 3
                vertices#[vcount,c-1]=
                ↳ValFloat(GetStringToken(text$," ",c+1))
            next c
        rem *** If it's a face, save vertex numbers ***
        elseif linetype$ = "f"
            inc fcount
            for c = 1 to 3
                faces[fcount,c-1] =
                ↳Val(GetStringToken(text$," /",c*3-1))
            next c
        endif
        rem *** Read next line ***
        text$ = ReadLine(fileid)
    endwhile
    rem *** Record vertex count in cell zero ***
    vertices#[0,0] = vcount
    rem *** Record faces count in cell zero ***
    faces[0,0] = fcount
    rem *** Close the file ***
    CloseFile(fileid)
endfunction

```

The required changes to `DrawWireframe()` are highlighted:

```

rem *** Draws wireframe of model ***
function DrawWireframe(scale#)
    rem *** FOR each face DO ***
    for face = 1 to faces[0,0]
        rem *** FOR each edge DO ***
        for edge = 0 to 2
            rem *** Get starting point ***
            x1# = GetScreenXFrom3D(vertices#[
            ↳faces[face,edge],0]*scale#,
            ↳vertices#[faces[face,edge],1]* scale#,
            ↳vertices#[faces[face,edge],2]*scale#)
            y1# = GetScreenYFrom3D(vertices#
            ↳[faces[face,edge],0]*scale#,
            ↳vertices#[faces[face,edge],1]*scale#,
            ↳vertices#[faces[face,edge],2]*scale#)
            rem *** Get end point ***
            fin = (edge+1) mod 3
            x2# = GetScreenXFrom3D(vertices#
            ↳[faces[face,fin],0]*scale#,
            ↳vertices#[faces[face,fin],1]*scale#,
            ↳vertices#[faces[face,fin],2]*scale#)
            y2# = GetScreenYFrom3D(vertices#
            ↳[faces[face,fin],0]*scale#,
            ↳vertices#[faces[face,fin],1]*scale#,
            ↳vertices#[faces[face,fin],2]*scale#)
            DrawLine(x1#,y1#,x2#,y2#,200,200,0)
        next edge
    next face
endfunction

```


Appendix A

ASCII Codes

First hex digit	Second hex digit															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	null							bell	back-space	H tab	line feed	V tab	form feed	return		
1												escape				
2	space	!	ì	#	\$	%	&	ë	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	ë	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	delete

ASCII characters occupy the 7 least-significant bits of a byte and are coded as 00 to 7F (hexadecimal).
The characters coded 0 to 1F are non-printing control characters. Only those which may affect cursor position or produce audio output have been named in the table given above.

Index

Symbols

^	73	algorithm	2
-	72	alpha channel	148, 459
..	314	ambient light	780
*	72	anchor	658
/	73	and	10, 102
//	47	anti-aliasing	147
&&	315	arithmetic expression	72
%	311	array concepts	271
+	72	array declaration	272
<	98	arrays	269, 893
<<	313	arrays and functions	296
<>	99	arrays of records	309
=	70, 99	arrays within records	897
>	99	Asc()	236
>>	314	ASCII	100
	316	ASin()	257
~~	316	ASinRad()	257
0C	311	assignment statement	65, 70
0X	311	ATan()	257
3D animation	782	ATanFull()	258
3D primitives	778	ATanFullRad()	258
3D ray casting	837	ATanRad()	257
3D axes	772	atlas texture image	451
#constant	68		
#include	88		
		B	
A		back buffer	81
Abs()	259	back face	777, 838
accelerometer	432	benchmarking	761
accessing array elements	272	billboarding	828
accessing files	324	Bin()	239
accessing memory	884	binary	311
ACos()	257	Binary Selection	4, 98
ACosRad()	257	bit	884
actual parameter	197	bitmap	146
adding to an array	285	bitwise and operator	315
AddParticlesColorKeyFrame()	350	bitwise boolean operators	314
AddParticlesForce()	348	bitwise exclusive or operator	316
AddSpriteShapeBox()	631	bitwise not operator	314
AddSpriteShapeCircle()	633	bitwise or operator	316
AddSpriteShapePolygon()	631	BMP	146
AddVirtualButton()	392	Boolean expression	4
AddVirtualJoystick()	421	bottom-up programming	219
AddZipEntry()	335	bytes	884
advertising	756		
AGK BASIC	36	C	
		C	36
		C++	36
		CalculateSpritePhysicsCOM()	627
		calculation	21
		calling a function	193
		camera	462, 783, 790, 816

Cartesian coordinates	250	CreatePulleyJoint2()	679
case	112	CreateRevoluteJoint()	666
Ceil()	260	CreateSprite()	149
Chr()	236	CreateText()	166
ClearLightDirectionals()	832	CreateWeldJoint()	658
ClearLightPoints()	835	CreateZip()	335
ClearParticlesColors()	352	cropping	600
ClearParticlesForces()	349	cull	777
ClearScreen()	605		
ClearSpriteShapes()	634	D	
CloneObject()	796	data	20, 64
CloneSprite()	154	data storage	304
CloseFile()	326	date	748
CloseZip()	336	dec	74
collisions	628, 837	declaring variables	304
colour channels	148	DecodeQRCode()	754
comments	47	default	112
comparison	21	degrees	256
compile button	43	Delete()	246
compiler	37	DeleteAdvert()	757
CompleteRawJoystickDetection()	437	DeleteAllImages()	156
complex number	919	DeleteAllSprites()	155
compound conditions	102	DeleteAllText()	169
computer program	3	DeleteEditBox()	418
condition	4, 98	DeleteFile()	331
conditional instruction	4	DeleteFolder()	334
controlling speed	523	DeleteImage()	156
coordinates	250	DeleteJoint()	662
CopyImage()	457	DeleteLightDirectional()	832
copyright issues	180	DeleteLightPoint()	835
Cos()	251, 254	DeleteMemblock()	890
CountStringTokens()	240	DeleteMusic()	162
CRB button	43	DeleteObject()	798
CreateAdvert()	756	DeletePhysicsForce()	640
CreateDistanceJoint()	663	DeleteSound()	157
CreateDummySprite()	652	DeleteSprite()	155
CreateEditBox()	405	DeleteText()	168
CreateGearJoint()	678	DeleteVirtualButton()	397
CreateImageFromMemBlock()	911	DeleteVirtualJoystick()	427
CreateLightDirectional()	789	deleting from an array	290
CreateLightPoint()	833	denary	311
CreateLineJoint()	676	depth buffer	863
CreateMemblock()	885	designing a function	192
CreateMemblockFromImage()	910	designing algorithms	2
CreateMouseJoint()	665	desk checking	29
CreateObjectBox()	793	device identity	442
CreateObjectCone()	794	directional light	780
CreateObjectCylinder()	795	distance joint	663
CreateObjectPlane()	796	do .. loop	135
CreateObjectSphere()	793	DrawSprite()	606
CreatePhysicsForce()	638	dry running	29
CreatePrismaticJoint()	673	dword	304

dynamic arrays	293	GetButtonPressed()	429
dynamic object	614	GetButtonReleased()	430
E		GetButtonState()	430
edge	776	GetCameraAngleX()	822
edit box	405	GetCameraAngleY()	822
else	106	GetCameraAngleZ()	822
elseif	111	GetCameraExists()	462
emitter	342	GetCameraQuatW()	871
EncodeQRCode()	753	GetCameraQuatX()	871
endif	98	GetCameraQuatY()	871
errors	759	GetCameraQuatZ()	871
executable file	37	GetCameraX()	822
exit	134	GetCameraY()	822
exitfunction	200	GetCameraZ()	822
ExtractZip()	336	GetCapturedImage()	463
F		GetChosenImage()	461
fields	305	GetContactSpriteID1()	644
FileEOF()	328	GetContactSpriteID2()	644
file handling	323	GetContactWorldX()	644
FileIsOpen()	330	GetContactWorldY()	644
file management	330	GetCurrentDate()	748
FinishPulleyJoint()	679	GetCurrentEditBox()	408
first person perspective	823	GetCurrentTime()	751
FixEditBoxToScreen()	589	GetDayFromUnix()	750
FixParticlesToScreen()	354	GetDayOfWeek()	748
FixSpriteToScreen()	589	GetDeviceHeight()	580
FixTextToScreen()	589	GetDeviceName()	442
float	80	GetDeviceWidth()	580
Floor()	260	GetDirectionAngle()	432
Fmod()	262	GetDirectionSpeed()	434
folder management	331	GetDirectionX()	433
font replacement	383	GetDirectionY()	433
for..endfor	14	GetDisplayAspect()	580
formal parameter	197	GetDrawingSetupTime()	761
for..next	128	GetDrawingTime()	761
FOV	816	GetEditBoxActive()	417
frame	81, 604	GetEditBoxChanged()	408
front buffer	81	GetEditBoxExists()	418
front face	777	GetEditBoxHasFocus()	408
functions	192	GetEditBoxHeight()	406
G		GetEditBoxLines()	411
game description	176	GetEditBoxText()	407
game design	176	GetEditBoxVisible()	417
gear join	678	GetEditBoxWidth()	406
Get3DVectorXFromScreen()	860	GetEditBoxX()	407
Get3DVectorYFromScreen()	860	GetEditBoxY()	407
Get3DVectorZFromScreen()	860	GetErrorOccurred()	759
GetAccelerometerExists()	432	GetFileExists()	330
		GetFileSize()	331
		GetFirstContact()	643
		GetFirstFile()	334
		GetFirstFolder()	333

GetFolder()	331	GetObjectRayCastBounceY()	850
GetFrameTime()	523	GetObjectRayCastBounceZ()	850
GetHoursFromUnix()	751	GetObjectRayCastDistance()	855
GetImage()	458, 606	GetObjectRayCastHitID()	848
GetImageExists()	450	GetObjectRayCastNormalX()	852
GetImageFilename()	459	GetObjectRayCastNormalY()	852
GetImageHeight()	451	GetObjectRayCastNormalZ()	852
GetImageWidth()	451	GetObjectRayCastNumHits()	848
GetJointExists()	682	GetObjectRayCastSlideX()	849
GetJointReactionForceX()	661	GetObjectRayCastSlideY()	849
GetJointReactionForceY()	661	GetObjectRayCastSlideZ()	849
GetJointReactionTorque()	682	GetObjectRayCastX()	842
GetJoystickX()	428	GetObjectRayCastY()	842
GetJoystickY()	428	GetObjectRayCastZ()	842
GetKeyboardExists()	440	GetObjectTransparency()	801
GetLastChar()	403	GetObjectVisible()	802
GetLastError()	759	GetObjectX()	810
GetLeapYear()	749	GetObjectY()	810
GetLightDirectionalExists()	832	GetObjectZ()	810
GetLightPointExists()	835	GetOrientation()	581
GetManagedSpriteCount()	761	GetParticleDrawnPointCount()	762
GetManagedSpriteDrawCalls()	762	GetParticleDrawnQuadCount()	762
GetManagedSpriteDrawnCount()	762	GetParticlesAngle()	356
GetManagedSpriteSortedCount()	762	GetParticlesAngleRad()	356
GetMemblockByte()	888	GetParticlesDepth()	356
GetMemblockExists()	890	GetParticlesDirectionX()	356
GetMemblockFloat()	888	GetParticlesDirectionY()	357
GetMemblockInt()	888	GetParticlesExists()	357
GetMemblockShort()	888	GetParticlesFrequency()	357
GetMemblockSize()	890	GetParticlesLife()	357
GetMilliSeconds()	83	GetParticlesMaxReached()	358
GetMinutesFromUnix()	751	GetParticlesSize()	358
GetMonthFromUnix()	750	GetParticlesVisible()	358
GetMouseExists()	435	GetParticlesX()	358
GetMultiTouchExists()	595	GetParticlesY()	359
GetMusicExists()	162	GetPhysicsCollision()	628
GetNextContact()	644	GetPhysicsCollisionWorldX()	629
GetNextFile()	335	GetPhysicsCollisionWorldY()	629
GetNextFolder()	333	GetPhysicsCollisionX()	629
GetObjectAngleX()	811	GetPhysicsCollisionY()	629
GetObjectAngleY()	811	GetPhysicsTime()	761
GetObjectAngleZ()	811	GetPixelsDrawn()	763
GetObjectCullMode()	803	GetPointerPressed()	163
GetObjectDepthReadMode()	866	GetPointerReleased()	163
GetObjectDepthWrite()	866	GetPointerState()	163
GetObjectExists()	798	GetPointerX()	164
GetObjectInScreen()	823	GetPointerY()	164
GetObjectQuatW()	870	GetRawAccelX()	434
GetObjectQuatX()	870	GetRawAccelY()	434
GetObjectQuatY()	870	GetRawAccelZ()	434
GetObjectQuatZ()	870	GetRawFirstTouchEvent()	596
GetObjectRayCastBounceX()	850	GetRawJoystickButtonPressed()	437

GetRawJoystickButtonReleased()	437	GetSpriteColorGreen()	475
GetRawJoystickButtonState()	438	GetSpriteColorRed()	475
GetRawJoystickExists()	437	GetSpriteContactSpriteID2()	646
GetRawJoystickRX()	438	GetSpriteContactWorldX()	647
GetRawJoystickRY()	438	GetSpriteContactWorldY()	647
GetRawJoystickRZ()	438	GetSpriteDepth()	154, 472
GetRawJoystickX()	438	GetSpriteDistance()	521
GetRawJoystickY()	438	GetSpriteExists()	471
GetRawJoystickZ()	438	GetSpriteFirstContact()	646
GetRawKeyPressed()	440	GetSpriteGroup()	506
GetRawKeyReleased()	440	GetSpriteHeight()	479
GetRawKeyState()	441	GetSpriteHit()	165, 476
GetRawLastKey()	441	GetSpriteHitCategory()	509
GetRawMouseLeftPressed()	435	GetSpriteHitGroup()	506
GetRawMouseLeftReleased()	435	GetSpriteHitTest()	475
GetRawMouseLeftState()	435	GetSpriteImageID()	479
GetRawMouseRightPressed()	435	GetSpriteNextContact()	647
GetRawMouseRightReleased()	435	GetSpritePhysicsAngularVelocity()	621
GetRawMouseRightState()	435	GetSpritePhysicsMass()	625
GetRawMouseX()	436	GetSpritePhysicsVelocityX()	616
GetRawMouseY()	436	GetSpritePhysicsVelocityY()	616
GetRawNextTouchEvent()	596	GetSpritePixelFromX()	491
GetRawTouchCount()	595	GetSpritePixelFromY()	493
GetRawTouchCurrentX()	598	GetSpriteVisible()	471
GetRawTouchCurrentY()	598	GetSpriteWidth()	479
GetRawTouchLastX()	598	GetSpriteX()	478
GetRawTouchLastY()	598	GetSpriteXByOffset()	496
GetRawTouchReleased()	597	GetSpriteXFromPixel()	493
GetRawTouchStartX()	598	GetSpriteY()	478
GetRawTouchStartY()	598	GetSpriteYByOffset()	496
GetRawTouchTime()	597	GetSpriteYFromPixel()	493
GetRawTouchType()	596	GetStringToken()	241
GetRawTouchValue()	599	GetTextAlpha()	367
GetRayCastFraction()	528	GetTextBlue()	367
GetRayCastNormalX()	528	GetTextCharAngle()	379
GetRayCastNormalY()	528	GetTextCharAngleRad()	379
GetRayCastSpriteID()	527	GetTextCharColorAlpha()	381
GetRayCastX()	527	GetTextCharColorBlue()	381
GetRayCastY()	527	GetTextCharColorGreen()	381
GetResumed()	766	GetTextCharColorRed()	381
GetScreenXFrom3D()	857	GetTextCharX()	377
GetScreenYFrom3D()	857	GetTextCharY()	377
GetSeconds()	83	GetTextDepth()	374
GetSecondsFromUnix()	751	GetTextExists()	374
GetSoundExists()	158	GetTextGreen()	367
GetSoundInstances()	158	GetTextHitTest()	374
GetSoundsPlaying()	158	GetTextInput()	401
GetSpriteAngle()	473	GetTextInputCancelled()	401
GetSpriteAngleRad()	473	GetTextInputCompleted()	401
GetSpriteCollision()	519	GetTextInputState()	402
GetSpriteColorAlpha()	475	GetTextLength()	371
GetSpriteColorBlue()	475	GetTextRed()	367

GetTextSize()	369	joints	658
GetTextTotalHeight()	369	joystick	421, 437
GetTextTotalWidth()	370	JPG	146
GetTextVisible()	369		
GetTextX()	368	K	
GetTextY()	368	keyboard	400, 440
GetUnixFromDate()	752	keyword	67
GetUpdateTime()	761	kinematic object	614
GetViewOffsetX()	592		
GetViewOffsetY()	592	L	
GetViewZoom()	590		
GetVirtualButtonExists()	397	Left()	234
GetVirtualButtonPressed()	396	left-handed coordinate system	772
GetVirtualButtonReleased()	396	Len()	232
GetVirtualButtonState()	397	levels of detail	22
GetVirtualHeight()	580	library	209
GetVirtualJoystickExists()	427	lighting	780
GetVirtualJoystickX()	424	lights	831
GetVirtualJoystickY()	424	line joint	676
GetVirtualWidth()	580	lines	774
GetWritePath()	331	little endian	887
GetYearFromUnix()	750	LoadImage()	149
global variables	215	LoadMusic()	160
gosub	207	LoadObject()	786
		LoadShader()	867
H		LoadSound()	156
hangman	6	LoadSubImage()	451
Hex()	240	local axes	779
		local variables	195
I		lossless formats	146
IDE	39	lossy formats	146
if	98	Lower()	234
if..then	107	M	
image formats	146	machine code	36
ImageJoiner utility	455	main.agc	36
images	146, 450	MakeFolder()	332
image transparency	147	MakeParticles()	342
imaginary numbers	919	Mandelbrot	918
inc	74	mapping a pixel	912
inertial axes	808	math functions	250
infinite loops	20	meaningful names	67
initialising arrays	272	memblock	884
input	21	Message()	51
Insert()	245	Mid()	235
InstanceObject()	797	Milkshape	773
integer	20, 64	mini-spec	193
integer variables	65	mips	3
IsCapturingImage()	463	mod	72
IsChoosingImage()	461	model resolution	778
iteration	3, 14, 124	modular software	211
J		monospaced font	383

motor	670	parentheses	76
mouse	435	particle emitter	342
mouse joint	665	particles	342
mouse scrolling	593	paused apps	766
MoveCameraLocalX()	822	PauseMusic()	162
MoveCameraLocalY()	822	physical joysticks	427
MoveCameraLocalZ()	822	physics	614
MoveObjectLocalX()	804	physics categories	648
MoveObjectLocalY()	804	physics groups	648
MoveObjectLocalZ()	804	PhysicsRayCast()	653
moving sprites	511	PhysicsRayCastCategory()	656
MP3 format,	180	PhysicsRayCastGroup()	655
multi-dimensional arrays	294	planes	775
multiple counts	276	PlayMusic()	160
multiple parameters	199	PlaySound()	157
multi-way selection	7, 10	PNG	55, 146
music	159	point light	780
mutually exclusive conditions	7	point lights	833
N			
named constants	68	polycount	778
naming rules	67	polygon	776
nested if statements	8, 109	polygonal mesh	777
nested loops	135	Pos()	242
nested parentheses	76	pre-conditions	200
nested records	307, 899	primitives	793
new project	56	Print()	45, 52, 77
normals	780	PrintC()	47
not	12, 105	PrintImage()	459
numbers to text	292	prismatic joint	673
number systems	311	program	3
O			
object code	37	program code	42
ObjectRayCast()	837	program data	64
Object Reflectivity	835	programming language	3
ObjectSphereCast()	843	project	36
ObjectSphereSlide()	848	proportional font	383
Occurs()	244	proportional fonts	456
offset lines	254	pulley joint	679
OGG Vorbis	180	Q	
one dimensional arrays	271	QR coding	753
OpenToRead()	326	Quaternion rotation	869
OpenToWrite()	324	R	
operator precedence	75	radians	256
or	11, 102	Random()	84
output	21	random non-repeating values	277
overall game document	179	RandomSign()	87
P			
parameters	196	range	817
		ray casting	524
		ReadFloat()	327
		ReadInteger()	327
		ReadLine()	328

ReadString()	327	sentinel	284
real	20, 64	sequence	3
real variables	66	SetAdvertPosition()	757
record	305	SetBorderColor()	58
records containing strings	903	SetCameraFOV()	816
record structure	894	SetCameraLookAt()	787
rem	47	SetCameraPosition()	787
remend	47	SetCameraRange()	817
remstart	47	SetCameraRotation()	820
Render()	605	SetCameraRotationQuat()	870
repeat .. until	17, 126	SetClearColor()	51
Replace()	248	SetDefaultMagFilter()	464
ResetParticleCount()	344	SetDefaultMinFilter()	464
ResetSpriteUV()	491	SetDisplayAspect()	57
ResetTimer()	83	SetEditBoxActive()	416
resources	146	SetEditBoxBackgroundColor()	413
ResumeMusic()	162	SetEditBoxBackgroundImage()	413
return	207	SetEditBoxBorderColor()	414
return types	201	SetEditBoxBorderImage()	414
revolute joint	666	SetEditBoxBorderSize()	415
Right()	235	SetEditBoxCursorBlinkTime()	415
roll	788	SetEditBoxCursorColor()	415
RotateCameraGlobalX()	821	SetEditBoxCursorWidth()	415
RotateCameraGlobalY()	821	SetEditBoxDepth()	418
RotateCameraGlobalZ()	821	SetEditBoxFocus()	407
RotateCameraLocalX()	821	SetEditBoxFontImage()	410
RotateCameraLocalY()	821	SetEditBoxMaxChars()	407
RotateCameraLocalZ()	821	SetEditBoxMaxLines()	411
RotateObjectGlobalX()	808	SetEditBoxMultiLine()	410
RotateObjectGlobalY()	809	SetEditBoxPosition()	406
RotateObjectGlobalZ()	809	SetEditBoxScissor()	416
RotateObjectLocalX()	807	SetEditBoxSize()	406
RotateObjectLocalY()	807	SetEditBoxText()	411
RotateObjectLocalZ()	807	SetEditBoxTextColor()	410
Round()	261	SetEditBoxTextSize()	409
routines	192	SetEditBoxVisible()	417
run	3	SetErrorMode()	759
Run button	43	SetFolder()	333
S		SetGenerateMipmaps()	466
		SetGlobal3DDepth()	863
		SetImageMagFilter()	464
		SetImageMask()	459
		SetImageMinFilter()	465
		SetImageWrapU()	486
		SetImageWrapV()	486
		SetInneractiveDetails()	756
		SetJointLimitOff()	673
		SetJointLimitOn()	672
SaveImage()	459	SetJointMotorOff()	670
screen buffers	604	SetJointMotorOn()	670
screen coordinates	583	SetJointMouseTarget()	665
ScreenFPS()	524	SetJoystickDeadZone()	429
screen handling	580		
screen layouts	176		
screen size	57		
ScreenToWorldX()	586		
ScreenToWorldY()	586		
scrolling	583		
select	112		
selection	3, 4		

SetJoystickScreenPosition()	427	SetPhysicsWallRight()	638
SetLightDirectionalColor()	831	SetPhysicsWallTop()	638
SetLightDirectionalDirection()	831	SetPrintColor()	48
SetLightPointColor()	834	SetPrintSize()	50
SetLightPointPosition()	833	SetPrintSpacing()	50
SetLightPointRadius()	834	SetRandomSeed()	86
SetMemblockByte()	886	SetRawJoystickDeadZone()	439
SetMemblockFloat()	888	SetRawMouseVisible()	436
SetMemblockInt()	888	SetRawTouchValue()	598
SetMemblockShort()	887	SetResolutionMode()	583
SetMusicFileVolume()	162	SetScissor()	600
SetMusicSystemVolume()	163	SetShaderConstantByName()	868
SetObjectCollisionMode()	842	SetSoundSystemVolume()	157
SetObjectColor()	798	SetSpriteAngle()	473
SetObjectCullMode()	802	SetSpriteAngleRad()	473
SetObjectDepthReadMode()	864	SetSpriteCategoryBit()	509
SetObjectDepthWrite()	865	SetSpriteCategoryBits()	508
SetObjectImage()	799	SetSpriteCollideBit()	652
SetObjectLightMode()	835	SetSpriteCollideBits()	650
SetObjectLookAt()	809	SetSpriteColor()	474
SetObjectPosition()	803	SetSpriteColorAlpha()	474
SetObjectRotation()	805	SetSpriteColorBlue()	474
SetObjectRotationQuat()	870	SetSpriteColorGreen()	474
SetObjectScale()	812	SetSpriteColorRed()	474
SetObjectShader()	867	SetSpriteDepth()	153
SetObjectTransparency()	799	SetSpriteFlip()	481
SetObjectVisible()	802	SetSpriteGroup()	505
SetOrientationAllowed()	581	SetSpriteImage()	479
SetParticleFrequency()	343	SetSpriteOffset()	494
SetParticlesActive()	355	SetSpritePhysicsAngularDamping()	619
SetParticlesAngle()	345	SetSpritePhysicsAngularImpulse()	620
SetParticlesAngleRad()	346	SetSpritePhysicsAngularVelocity()	618
SetParticlesColorInterpolation()	351	SetSpritePhysicsCanRotate()	621
SetParticlesDepth()	354	SetSpritePhysicsCOM()	626
SetParticlesDirection()	346	SetSpritePhysicsDamping()	626
SetParticlesImage()	352	SetSpritePhysicsDelete()	617
SetParticlesLife()	344	SetSpritePhysicsForce()	621
SetParticlesMax()	344	SetSpritePhysicsFriction()	625
SetParticlesSize()	343	SetSpritePhysicsImpulse()	624
SetParticlesStartZone()	349	SetSpritePhysicsIsBullet()	629
SetParticlesVelocityRange()	345	SetSpritePhysicsIsSensor()	627
SetParticlesVisible()	354	SetSpritePhysicsMass()	625
SetPhysicsDebugOff()	630	SetSpritePhysicsOff()	617
SetPhysicsDebugOn()	630	SetSpritePhysicsOn()	614
SetPhysicsForcePosition()	641	SetSpritePhysicsRestitution()	617
SetPhysicsForcePower()	641	SetSpritePhysicsTorque()	619
SetPhysicsForceRange()	641	SetSpritePhysicsVelocity()	615
SetPhysicsGravity()	636	SetSpritePosition()	152
SetPhysicsMaxPolygonPoints()	634	SetSpritePositionByOffset()	496
SetPhysicsScale()	636	SetSpriteScale()	472
SetPhysicsWallBottom()	638	SetSpriteScaleByOffset()	498
SetPhysicsWallLeft()	638	SetSpriteScissor()	494

SetSpriteShape()	500	SetViewZoomMode()	587
SetSpriteShapeBox()	501	SetVirtualButtonActive()	394
SetSpriteShapeCircle()	502	SetVirtualButtonAlpha()	393
SetSpriteShapePolygon()	502	SetVirtualButtonColor()	393
SetSpriteSize()	151	SetVirtualButtonImageDown()	395
SetSpriteSnap()	493	SetVirtualButtonImageUp()	395
SetSpriteTransparency()	480	SetVirtualButtonPosition()	394
SetSpriteUV()	489	SetVirtualButtonSize()	394
SetSpriteUVBorder()	488	SetVirtualButtonText()	392
SetSpriteUVOffset()	485	SetVirtualButtonVisible()	394
SetSpriteUVScale()	483	SetVirtualJoystickActive()	425
SetSpriteVisible()	155	SetVirtualJoystickAlpha()	422
SetSpriteX()	477	SetVirtualJoystickDeadZone()	425
SetSpriteY()	478	SetVirtualJoystickImageInner()	423
SetSyncRate()	524	SetVirtualJoystickImageOuter()	423
SetTextAlignment()	372	SetVirtualJoystickPosition()	421
SetTextAlpha()	367	SetVirtualJoystickSize()	422
SetTextBlue()	367	SetVirtualJoystickVisible()	426
SetTextCharAngle()	377	SetVirtualResolution()	58
SetTextCharAngleRad()	377	SetWindowTitle()	59
SetTextCharColor()	380	shaders	783, 866
SetTextCharColorAlpha()	380	shift left operator	313
SetTextCharColorBlue()	380	shift operators	312
SetTextCharColorGreen()	380	shift right operator	314
SetTextCharColorRed()	380	ShowChooseImageScreen()	461
SetTextCharPosition()	375	ShowImageCaptureScreen()	462
SetTextCharX()	376	shuffling	279
SetTextCharY()	376	Sin()	253, 254
SetTextColor()	166	Sleep()	84
SetTextDefaultExtendedFontImage()	385	smallest value	133
SetTextDefaultFontImage()	383	software	3
SetTextDefaultMagFilter()	382	sound	156
SetTextDefaultMinFilter()	382	source code	37
SetTextDepth()	373	Space()	239
SetTextFontImage()	385	splash screen	55
SetTextGreen()	367	spot the difference game	175
SetTextInputMaxChars()	403	sprite and 3d objects	863
SetTextLineSpacing()	373	sprite bounding areas	499
SetTextMaxWidth()	370	sprite groups	505
SetTextPosition()	167	SpriteRayCast()	525
SetTextRed()	367	SpriteRayCastSingle()	532
SetTextScissor()	371	sprites	470
SetTextSize()	167	Sqrt()	259
SetTextSpacing()	372	starting AGK	39
SetTextString()	168	StartTextInput()	400
SetTextVisible()	168	state-transition diagram	180
SetTextX()	368	static object	614
SetTextY()	368	step	129
SetTransitionMode()	582	stepwise refinement	23, 181
setup.agc	54	StopMusic()	162
SetViewOffset()	591	StopSound()	157
SetViewZoom()	586	StopTextInput()	402

storing a character	891	UpdateParticles()	355
storing a string	891	Upper()	233
Str()	237	user-defined functions	191
string	20, 64	user input	87
string functions	232	user interaction	163
string operations	77	UV coordinates	781
string variables	66		
structured English	36, 101, 181	V	
structure diagrams	221	Val()	238
subimages	451	variable range	77
subimages.txt	452	variables	64
subroutines	192, 207	vectors	775
subscript	273	velocity	615
surface normals	780	vertex normals	780
Swap()	605	vertices	776
Sync()	81, 604	virtual buttons	392
syntax	37	virtual joystick	421
syntax diagram	45	virtual resolution	58
syntax error	37		
		W	
T		weld joint	658
Tan()	255	while .. endwhile	18, 124
testing iterative code	137	world coordinates	583
testing selective code	115	WorldToScreenX()	586
testing sequential code	91	WorldToScreenY()	586
text	366	WriteFloat()	325
text input	400	WriteInteger()	325
text resources	165	WriteLine()	325
texturing	781	WriteString()	325
time	748		
timer()	79	X	
tip of the day	39	x displacement	774
token	45	XY plane	776
top-down programming	212	XZ plane	776
torque	619		
touch scroll	599	Y	
touch statements	595	y displacement	774
touch zoom	599	YZ plane	776
trace table	27		
transforming 3D objects	803	Z	
translation process	37	z-axis	772
trigonometric functions	251	z displacement	774
Trunc()	261	Zip files	335
type	305	zooming	583, 816, 927
types of data	20		
U			
undim	294		
unit vector	775		
Unix	749		
Update()	604		

Also Available

from our website www.digital-skills.co.uk

Hands On DarkBASIC Pro Volume 1 742 pages (Printed or ebook PDF format)

Contents include:

Background Concepts	Video Cards and the Screen
Starting DarkBASIC Pro	File Handling
Data	Handling Music Files
Selection	Displaying Video Files
Iteration	Accessing the Keyboard
Drawing Statements	Mathematical Functions
Modular Programming	Images
String Functions	Sprites
The Game of Hangman	Sound
Arrays	2D Vectors
The Game of Bull and Touch	Two-Player Space Duel Game
Advanced Data Types and Operators	Using the Mouse
Bitmaps	Using a Joystick

Hands On DarkBASIC Pro Volume 2 726 pages (Printed or ebook PDF format)

Contents include:

3D Concepts and Terminology	BSP Models
3D Primitives	Elevators - Game
Texturing	Creating Terrain
Cameras	Landscape Matrices
Lighting	Manipulating Vertices
Meshes and Limbs	Pointers and Memblocks
Importing 3D Models	Shaders
User Control	ODE
Solitaire - The Board Game	Maths: Vectors and Matrices
Advanced Lighting and Texturing	Network Program
Collisions	FTP
Particles	Using DLLs

Hands On Milkshape 338 pages

(Printed or Ebook PDF format)

Contents include:

- Background Concepts
- Milkshape Basic Controls
- Principles of 3D Construction
- Vertices, Edges and Faces
- 3D Primitives
- Manipulating Vertices
- Reshaping Meshes
- Extrusion
- Using Milkshapes Additional Tools
- Groups
- Creating Models of Real World Objects
- Texturing Your Model
- Animation
- Exporting Your Model to DBPro

