

Hands On AGK BASIC

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

Alistair Stewart



Digital Skills

This is an extract from the book

Hands On AGK BASIC

by Alistair Stewart

You can purchase the complete publication (approx 900 pages) in either printed or ebook format from

The Games Creators [printed version only]

<http://www.thegamecreators.com/>

or

Digital Skills [printed or ebook (PDF)]

www.digital-skills.co.uk

Other books by Alistair Stewart include:

Hands On DarkBASIC Pro Volume 1

Hands On DarkBASIC Pro Volume 2

Hands On Milkshape

“Hands On AGK BASIC allowed me to jump right into the AGK programming language and create dFenz in just a few short weeks”

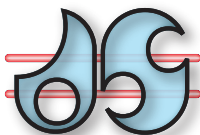
-Sean Mann, Napland Games



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

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
Appendix A - ASCII Codes	883
Index	884

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

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/AGKDownloads.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:

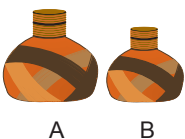
- One instruction per line
- Each instruction is unambiguous
- Each instruction is as brief as possible

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
- Pack in soil around base of shrub

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

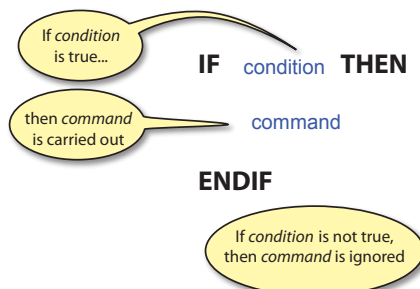
Notice that when we write a sentence containing the word IF, it consists of two main components:

- 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 options 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:

```
    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
```

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.

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.

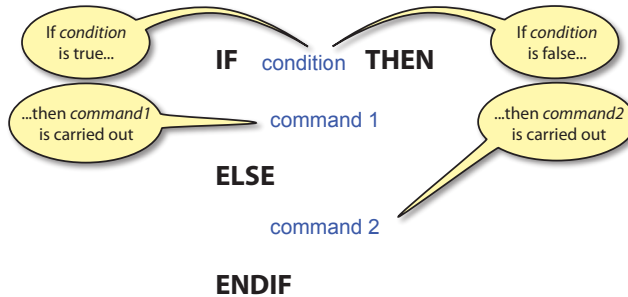
For instance, in our earlier example of playing a guessing game with a child, nothing happened if the child guessed wrongly. If the person holding the sweet were to eat it when the child's guess was incorrect, we could describe this setup with the following statement:

```
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

Because the IF statement (with or without the ELSE option) always offers two alternative options, the structure is known as **binary selection**.

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
```


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:

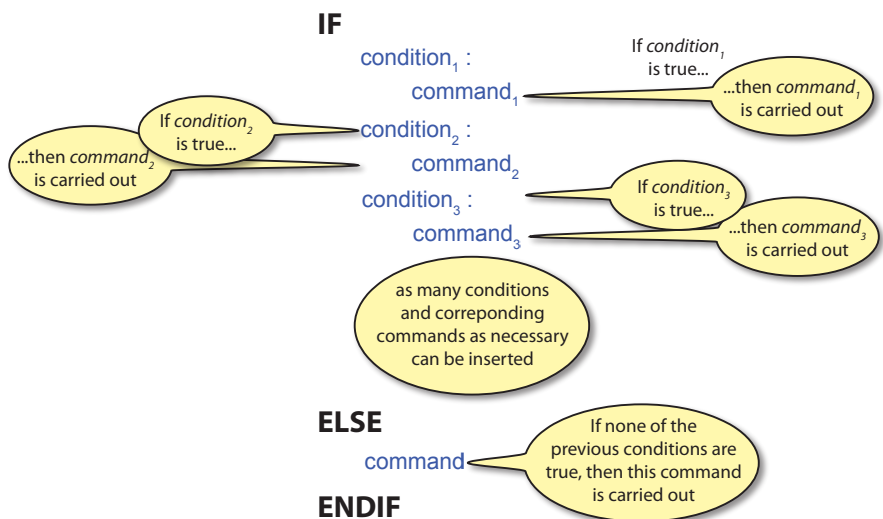
```
IF
  command is Go East:
    Move player's character eastward
  command is Go West:
    Move player's character westward
  command is Go North:
    Move player's character northward
  command is Go South:
    Move player's character southward
ELSE
  Display an error message
ENDIF
```

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

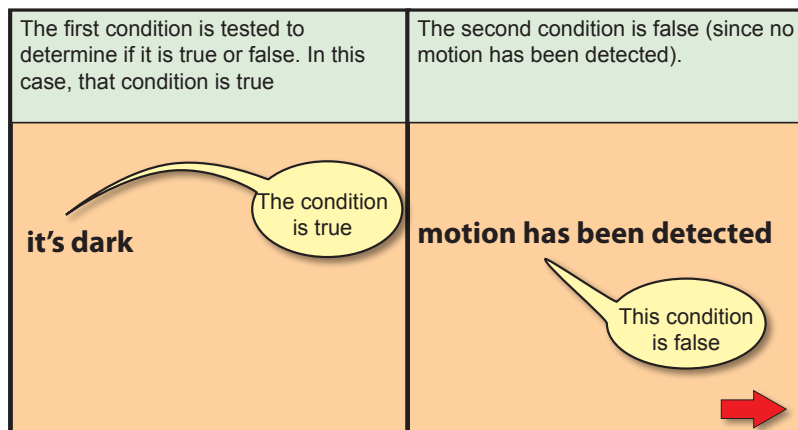


FIG-1.4

(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.
<pre> IF true AND false THEN Switch on light ENDIF </pre>	<pre> IF false THEN Switch on light ENDIF </pre> <p>The compound condition's final value is <i>false</i> so...</p> <p>...command not executed</p>

With two conditions there are four possible combinations of results. 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 the first line of 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:

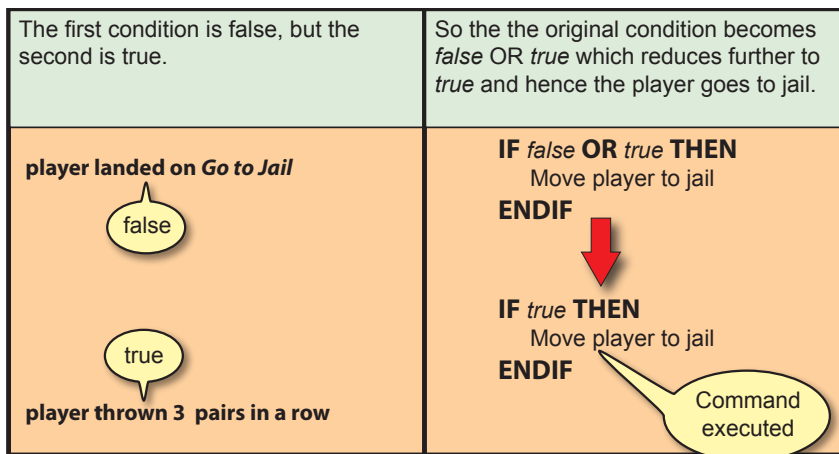
When OR is used, only one of the conditions involved needs to be true for the overall result to be true. Hence the results are determined by the following rules:

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

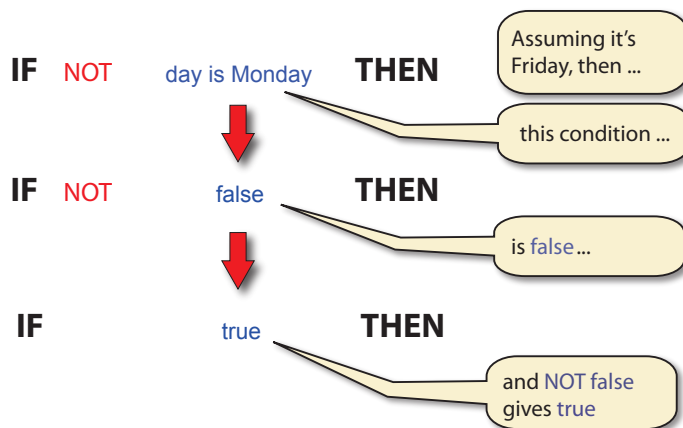
When NOT is used, the result given by the original condition (the bit without the NOT) is reversed. Hence the results are determined by the following rules:

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

For example, if we test for it not being Monday on a Friday, 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

♣The term **dice** is used for both singular and plural forms.

There are certain circumstances where 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 two parts to this statement. 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 but with numbers in the range 1 to 10.

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.

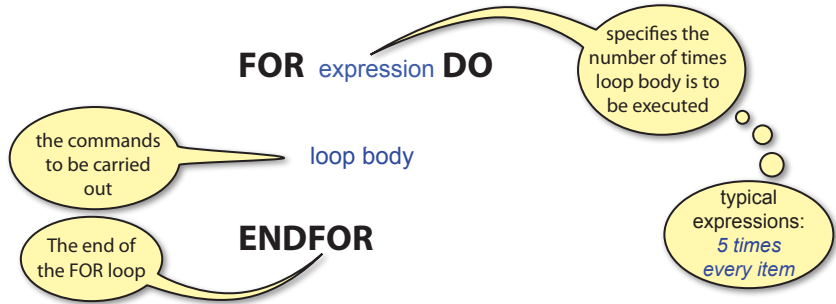
Write down a set of instructions which performs the task described above. Your solution should include the statements

```
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 1.16 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
```

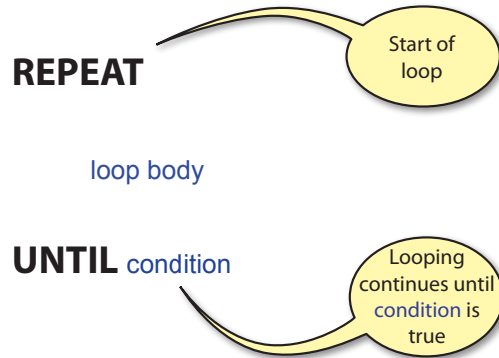
We could also use the REPEAT..UNTIL loop to describe how a slot machine (one-armed bandit) is played:

```
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 Blackjack 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 another card by saying "hit". One player is designated the dealer. The dealer must twist while his cards have a total value of less than 17. 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 17
```

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 17 DO
    Take another card
```

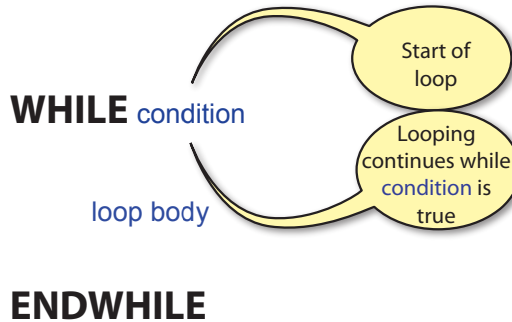
```
    Add new card's value to sum
ENDWHILE
```

Now determining if the sum is less than 17 is performed before the *Take another card* instruction. If the dealer's two cards already add up to 17 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 :

```
    a name
and  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 a floating point number or simply float) 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 setup 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 `->` 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 -> 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"
    ENDF
  ENDF
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"
    ENDF
  ENDF
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 set up a digital camcorder ready for future recordings 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 the traditional way of making a cup of British 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		A	B	C	D	E	F	G	H	I	J	
1											1											O
2											2											
3				A	A	A	A				3								O			
4									S		4											
5	C	C							D		5											
6				S					D		6		X	X	X							
7		D	D	D					D		7									O		
8						C			S		8											
9		S				C					9											
10				C	C						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
14. ENDIF
14. 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 number guess	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 <i>number guess</i>	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 <i>number guess</i>	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 <i>number guess</i>	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 <i>number guess</i>	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 <i>number guess</i>	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.

Since statement 13 contains a condition which is false, we return to statement 2 and

FIG-1.22

Working through a
Trace 6

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

then onto 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		Output
			number	guess	
1			42		Too high
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			
9					
10					
11					
12					
13	guess = number	F			
2				15	
3					

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 list 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 n :
    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	-

➤ 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 \rightarrow is used to assign a value to a data item when writing in a more program-like style. 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.
- 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           A = 5 litres  B = 0 litres
Fill B from A    A = 2         B = 3
Empty B          A = 2         B = 0
Empty A into B   A = 0         B = 2
Fill A           A = 5         B = 2
Fill B from A    A = 4         B = 3
```

A second solution is:

```
Fill B           A = 0         B = 3
Pour B into A    A = 3         B = 0
Fill B           A = 3         B = 3
Fill A from B    A = 5         B = 1
Empty A          A = 0         B = 1
Pour B into A    A = 1         B = 0
Fill B           A = 1         B = 3
Pour B into A    A = 4         B = 0
```

Activity 1.3

- An algorithm
- A computer program
- 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

- Sequence
Selection
Iteration
- Boolean expression
- Binary selection Multi-way selection
- No more than one of the conditions can be true at any given time.
- Boolean operators are: AND, OR, and NOT.
- A conditional statement is a statement which is executed only if a given set of conditions are met.
- 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
 - ENDWHILE
- Mark position in second grid with 0

The expansion of statement 5 would become:

- REPEAT
 - Get other player's call
 - IF other player's call matches position of ship THEN
 - Call HIT
 - ELSE
 - Call MISS
 - ENDIF
 - 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			15	
3				
4	guess = number	F		
6				
7	guess < number	T		
8				Too low
11				
12				
13	guess = number	F		
2			42	
3				
4	guess = number	T		
5				Correct
11				
12				
13	guess = number	T		

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).

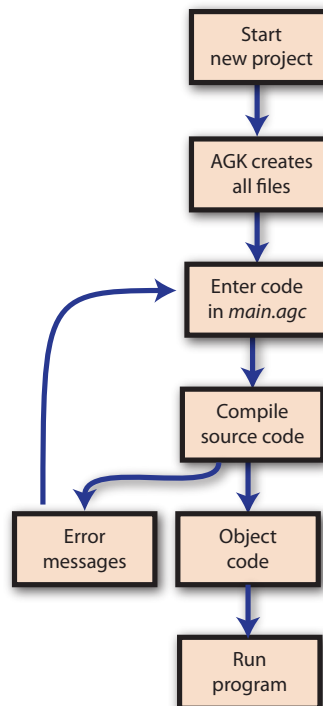
The compiler is a very exacting task master. The structure, or **syntax**, of every statement must be exactly right. If you make the slightest mistake, even something as simple as missing out a comma or misspelling a word, the translation process will fail. When this happens in AGK, a window appears 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?

Summary

- To program a computer, our structured English algorithms must be translated into a computer language.
- An AGK project consists of several files.
- Each AGK project is automatically assigned its own folder.
- The AGK BASIC code is known as the source code.
- The computer can only execute statements given in machine code.
- A compiler is used to translate a program from source code to machine code.
- Source code statements must conform to an exact grammar or syntax.
- Any deviation from that grammar is known as a syntax 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.

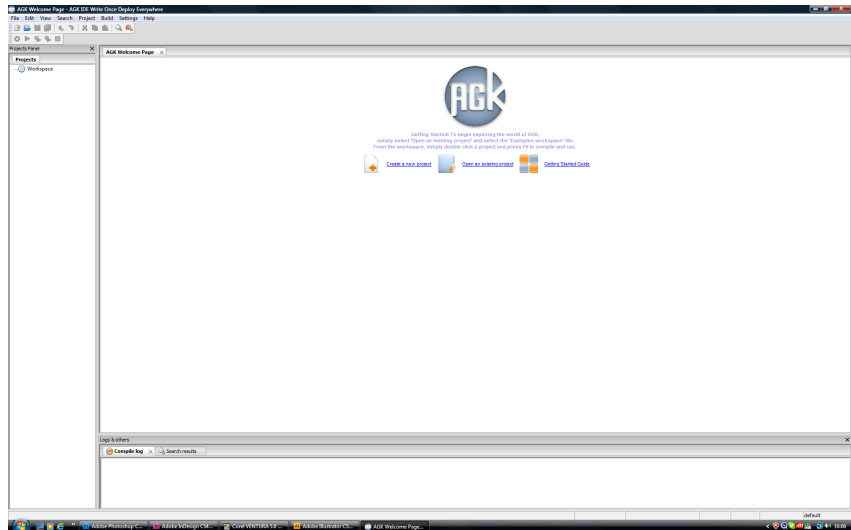
AGK was created by Lee Bamber, CEO of The Game Creators Ltd and was derived from his earlier creation, DarkBASIC which is a programming language designed to develop games for the PC platform only.

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

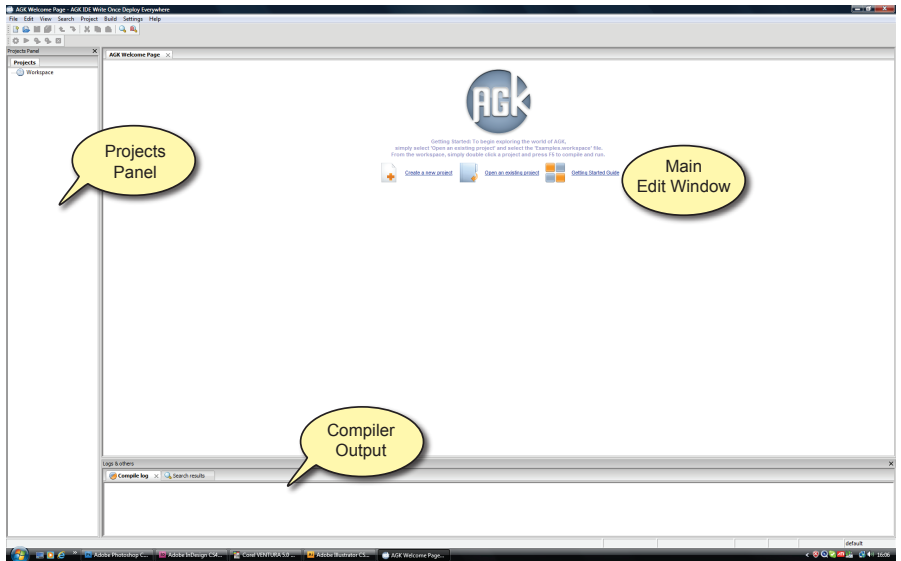


At the centre of the application window is the **Tip of the Day** window. If you don't want this to appear every time you start up AGK, just deselect the *Show tips at startup* check box. Once you close the **Tip of the Day** window, you are left with the three main areas of the AGK IDE (see FIG-2.3):

- 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 *Logs and others*) 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

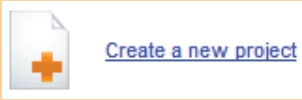
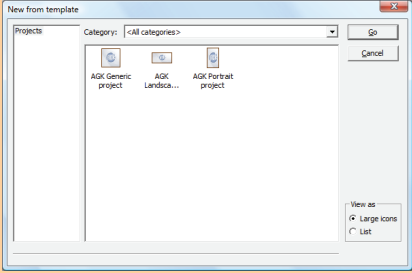
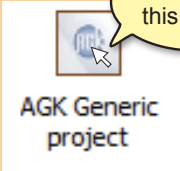
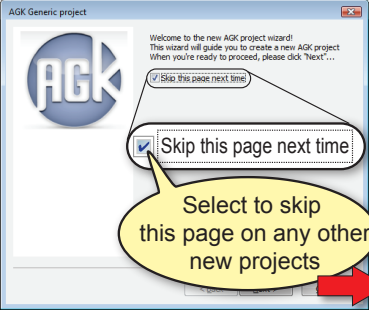
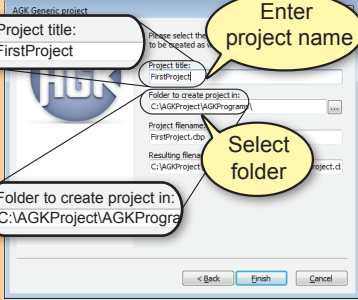
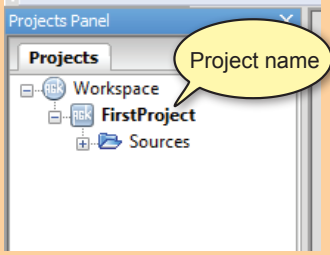
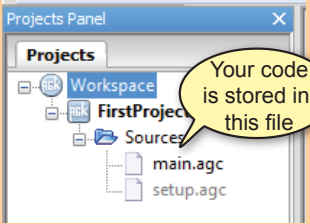
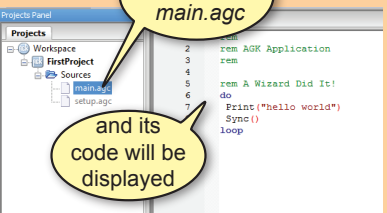

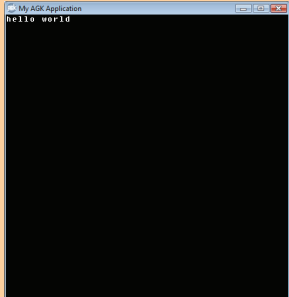
<p>Since this is the our first project, we click on the Create a new Project option in the Main Edit Window (File New Project would work too).</p>	<p>This displays the Create from Template window which offers three different layout styles for your new project's display.</p>
	
<p>For this project, AGK 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>
	

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. <i>main.agc</i> will contain your code.</p>	<p>Double clicking on <i>main.agc</i> 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 <i>main.agc</i>. 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.

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.

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

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.

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 even 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 File** 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.

Transferring Your App to a Tablet or Smartphone

Once your app has been coded and tested, it needs to be transferred wirelessly to the device you want to use it on.

To transfer the program to another device you need to broadcast a Wi-Fi signal. That means you either need to have a Wi-Fi router attached to your PC or be using a laptop with built-in Wi-Fi.

The device receiving your program must be running the **AGK Player** app. This app will detect your program being broadcast, download, and run it. Downloading the free AGK player app is straightforward for most platforms; just go to the app store and search for “AGK player”.

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 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.

To transfer the app you have written, start by running the AGK Player app on the target device (iPad, iPhone, Android tablet, etc.) and then press the **Compile and Broadcast** button on the AGK BASIC IDE to broadcast the app via your wireless router to your device. Your app should now download and automatically start executing on your device from within the app player.



Compile and Broadcast Button

If you want the app to run on your PC and smart device at the same time, press the **Compile, Run and Broadcast** button to the right of the **Compile and Broadcast** button.

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 **CB** 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 via the app player. To create a true app it has to be available from the app store for your device (see the Game Creators' web site for details).

Summary

➤ To start a new project:

Click on the *Start a New Project* option in AGK's IDE.
(Alternatively, choose **File|New|Project** from the main menu.)
Select *AGK Generic Project*.
Enter the project's name.
If necessary, select a new folder for the project.
In the **Projects Panel**, click on *Sources*.
Click on *main.agc* to display the project code.

➤ To execute a project:

Click on the **Compile** button.
Click on the **Run** button.

➤ To save an updated program:

Select **File|Save File**

➤ To run an app on your mobile device:

Run the AGK Player app on your mobile device.
Press the **Compile and Broadcast** button in the AGK IDE.

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 some 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 round-cornered 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:

```
Print 36           (parentheses are missing)
Print(Goodbye)    (no quotes)
Print('Help!')   (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)`

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

File|Save File

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, the line

```
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")
Sync()
```

are included in a program, they will create the output

```
Hello
Goodbye
```

Activity 2.6

Modify your program so that the code now reads

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

Compile and run the program. Resave your project.

Each message is on a separate line because the `Print()` statement always displays a

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

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 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.
remstart
```

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")
Sync()
```

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

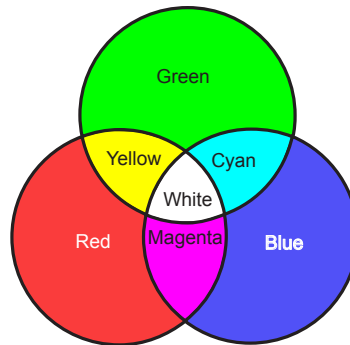
Other statements allow us to make various changes to how the text appearing on our screen is presented. We can change its colour, size, transparency and even the space between the characters.

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 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.

The statement's format is shown in FIG-2.12.

FIG-2.12

SetPrintColor()

```
SetPrintColor ( [ired] , [igreen] , [ibblue] [ , [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. (0: no green, 255: full green)
iblue	is an integer value (0 to 255) giving the strength of the blue component. (0: no blue, 255: full blue)
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()

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

FIG-2.13

SetPrintSize()

`SetPrintSize ((fsize))`

where:

fsize 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 ((fgap))`

where:

fgap 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.

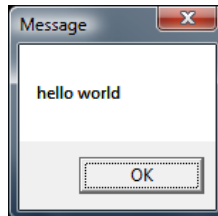
For example, the line

```
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.

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 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.

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?

As you can see from the output, for a simple program such as this, moving the statements has had no obvious effect on the display 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.

Summary

- The main file in an AGK project is *main.agk* 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 `SetColor()` to set a background colour for the app screen.

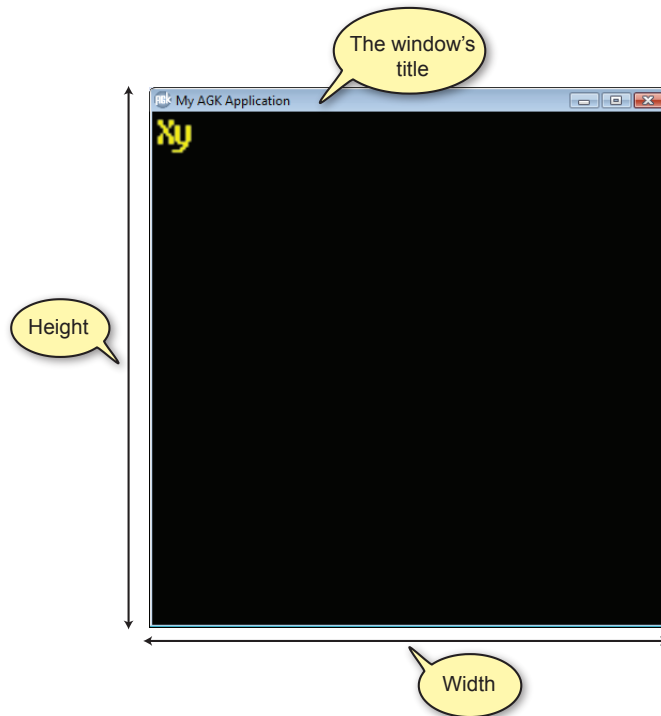
The Second Source File

Every project you create actually contains a second *.agc* file called *setup.agc*. 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. 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.

►► The splash screen only appears automatically when running your app on a PC.

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

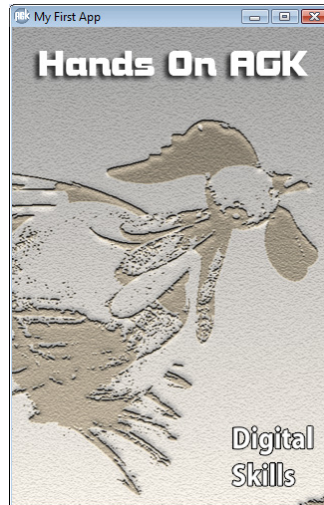
AGK Project's Files

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 *media* folder is where we need to place our splash screen graphic. 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/Media* 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.

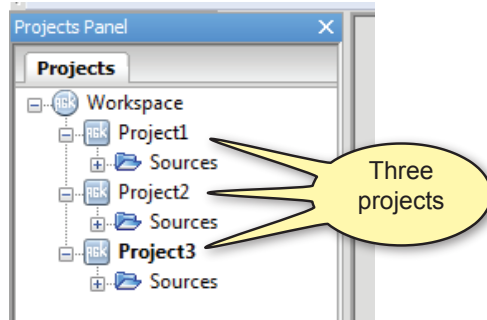
Starting a New Project

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

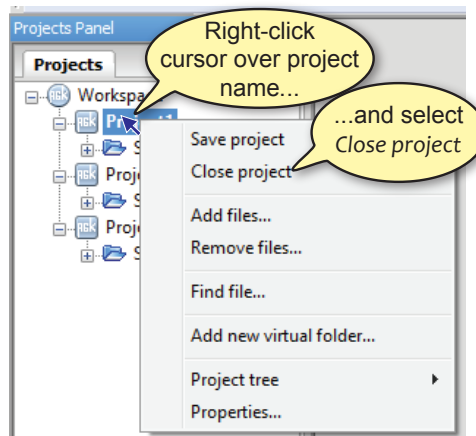
Multiple Projects



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.

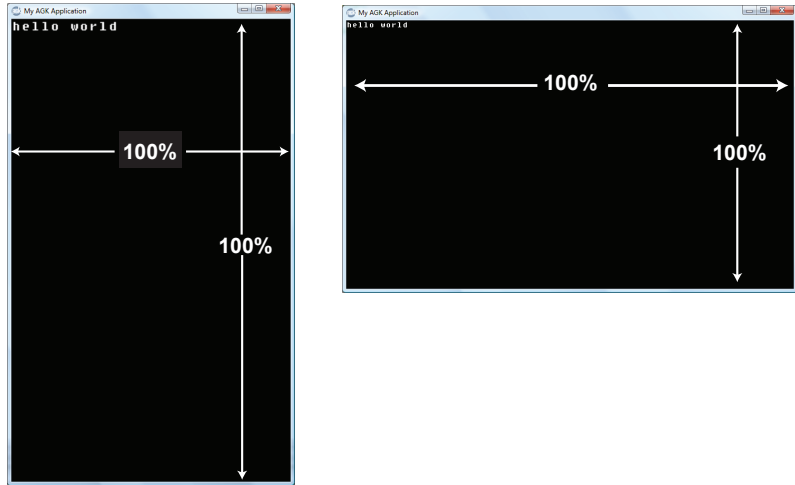
App Window Properties

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 height 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 ((fratio))

where:

fratio 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 *fratio* 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 320x480 app on a newer iPhone with its 640x960 resolution, your AGK will automatically expand to fill the full screen with every virtual pixel equal to two actual pixels.

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.

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

No matter whether you use the percentage or virtual pixel system, a problem arises when the device on which your app is running has a different aspect ratio (width -to- height) than that specified in your app. For example, while all of Apple's mobile devices have a width to height ratio of 3-to-4, the Asus EEE Transformer has an aspect ratio of 5-to-8 (resolution of 800 x 1280 in portrait mode). Expanding an app designed for a 3-to-4 screen to fill a 5-to-8 screen would cause distortion of any images being displayed (circles would become ovals!). AGK handles the problem of running a 3-to-4 app on a 5-to-8 screen by showing the app within the largest possible 3-to-4 area of the screen and adding a border colour to the unused part of the display.

SetBorderColor()

You can specify the border colour to be used when your 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 string 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

- In the *setup.agc* file you can specify the dimensions and window title for you app when running it in a Microsoft Windows environment.
- When running your app in Windows, a splash screen will appear while the app is loading if you have a file called *AGKSplash.png* in the project's *media* folder.
- By default, AGK uses a percentage coordinate system within the app window.
- To open further projects, use **File|New|Project** in the IDE.
- By default, all programs use a percentage system for screen dimensions.
- 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 not 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 File**

Activity 2.4

No solution required.

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)
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)
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 simple 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, an integer variable cannot store a string value.

Integer Variables

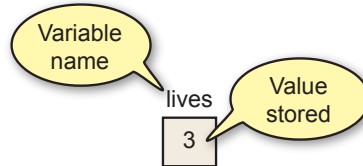
In AGK BASIC variables are created automatically as soon as we mention them in our code. For example, let's assume we want to store the number of lives allocated to a game player in a variable called *lives*. To do this in AGK BASIC we simply write the line:

```
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*).

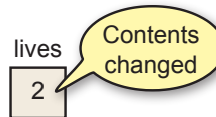
You are free to change the contents of a variable at any time by assigning it a different value. For example, we can change the contents of *lives* with a line such as:

```
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 valid

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

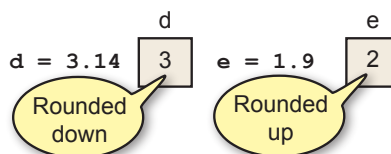
but the lines below will cause problems

```
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

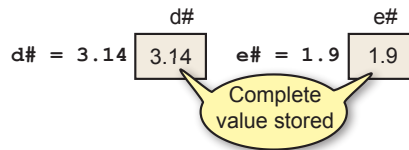
If you want to create a variable capable of storing a real number, then you must end the variable name with the hash (#) symbol. For example, if we write

```
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 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. A last consideration is that a real variable requires more space in the computer's memory than an integer one.

String Variables

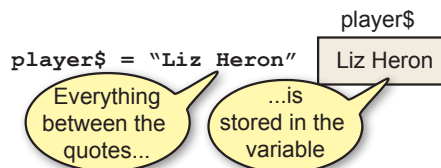
Finally, if you want to store a string value, you need to use a string variable. String variable names must end with a dollar (\$) sign. The value to be stored must be enclosed in double quotes. We could create a string variable named *player\$* and store the name *Liz Heron* in it using the statement:

```
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 *** An empty string ***
```

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

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.

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

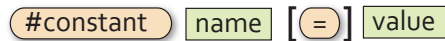
Which of the following are invalid variable names:

- | | | |
|----------|----------------|------------|
| 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`

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, the value 1000 can be assigned the name `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 modified 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.
- Names used for constants cannot end with \$ or #.

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

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
```

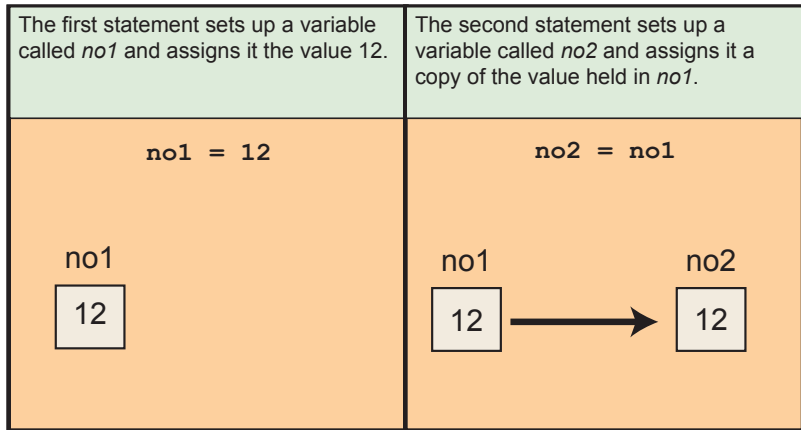
we can copy the contents of that variable into another variable with a line such as:

```
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 normally 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) `no2 = no1` b) `no3 = weight#` c) `result = description$`
d) `ans# = no1` e) `abc$ = weight#` f) `m# = description$`

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

You should already be familiar with most of these operators. 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`

► An **address bus** is a connection between the microprocessor and the memory. The address of the memory location to be accessed is transmitted along this bus.

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 subtracting 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 the `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 standard assignment approach.

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

FIG-3.10

The `inc` Statement

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

where:

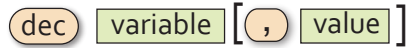
variable is the variable whose value is to be incremented.

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



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 ?`

When an expression contains more than one operator from the group multiplication, division, and remainder, these are also performed on a left-to-right basis.

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 running program stops unexpectedly at this point.

► When a program stops unexpectedly because of some error situation, this is known as a **runtime error**.

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 `Print()` (in combination with `Sync()`) can be used to display values on the screen using lines such as:

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

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 DC"
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 DC
```

Remember to close your old project.

➡ 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 several timer-related statements. One gives you access to the time your program has been running to the nearest second, another 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 us 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()
rem *** Display time ***
PrintC("Time elapsed : ")
Print(time_elapsed#)
Sync()
do
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 six lines so they are positioned between `do` and `loop`. 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.

What you see on the screen at any one instant is known as a **frame**. Every time a `Sync()` statement is executed a new frame is displayed on the screen. AGK reserves two areas of memory to handle the screen display. The first area holds the image currently displayed on the screen and is known as the **front buffer**. The second area holds the image of the next screen frame to be displayed and is known as the **back buffer**.

When a `Print()` or `PrintC()` statement is executed, the text to be displayed is copied into the 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 appear on the screen; what was the front buffer becomes the new back buffer and its contents are cleared.

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 its `PrintC()` statement back to its original position before `do`, 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

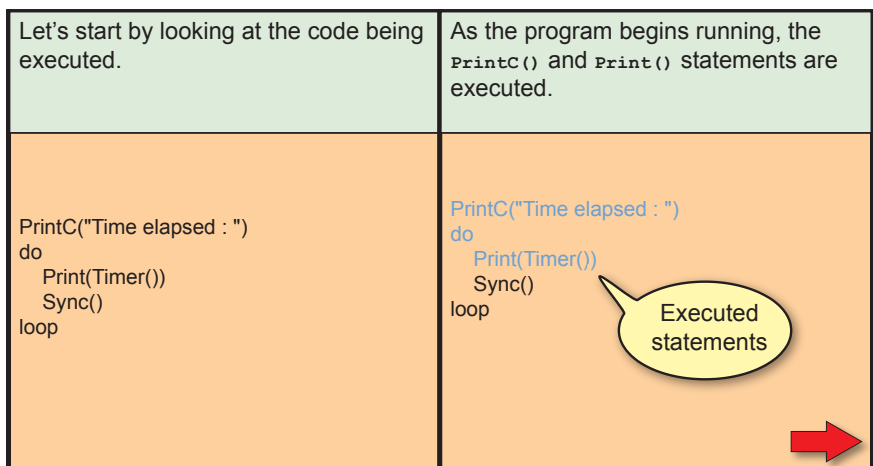
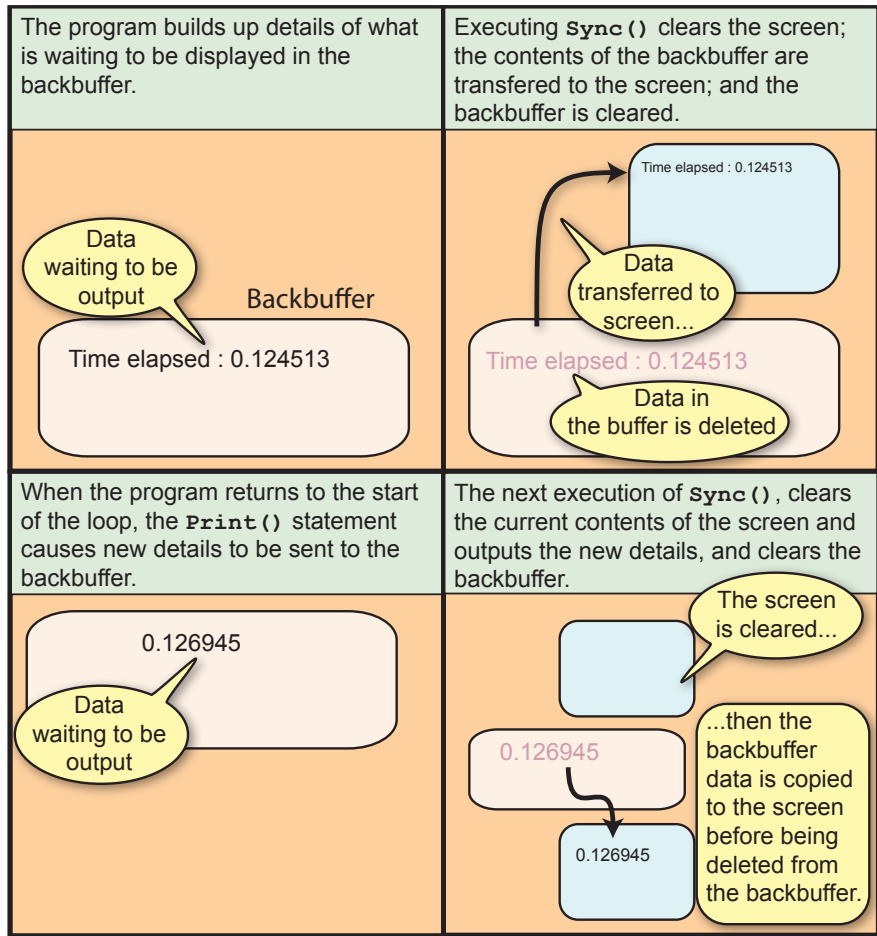


FIG-3.15

(continued)

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

```
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
```

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

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

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

FIG-3.16

Displaying Time
Elapsed in Minutes
and Seconds

```

rem *** Display time elapsed in ***
rem *** 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

ResetTimer()

```
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

GetMilliseconds()

```
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()` 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()`. 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 value giving the lowest value allowed.

ito is an integer value 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**.

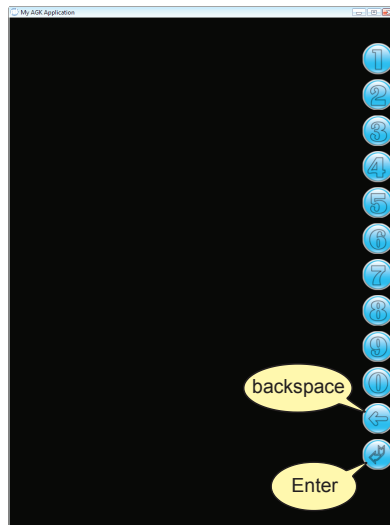
SetUpButtons() This function sets up 12 round buttons on the right of the app window. The buttons are labelled 0 to 9, `←` (*backspace*) and `↵` (*Enter*).

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

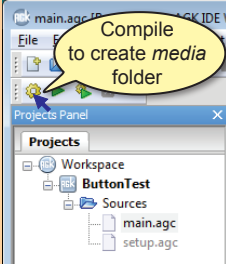
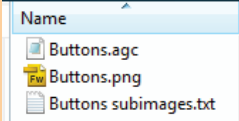
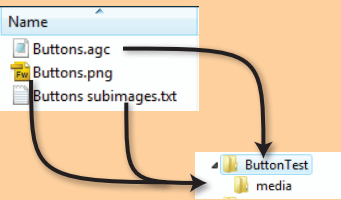
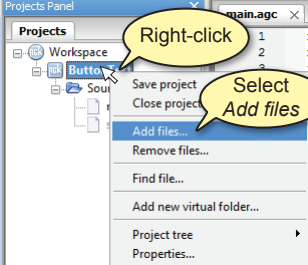
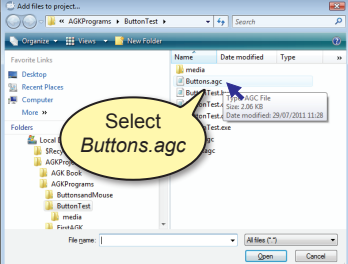
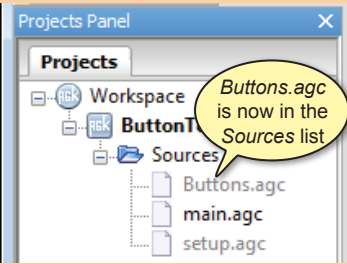
Buttons Layout



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>
	
<p>Double-click on the <i>Buttons.agc</i> file...</p>	<p>...to add the selected file to the <i>Sources</i> 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>
<pre>#include "Buttons.agc"</pre>	<pre>#include "Buttons.agc" SetUpButtons() value_entered = GetButtonEntry() PrintC("You entered ") Print(value_entered) Sync() do 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()
rem *** Display the value entered ***
PrintC("You entered ")
Print(value_entered)
Sync()
do
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-3.27.

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.

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
```

Activity 3.27

Reload your *Dice* program.

Make the necessary adjustments 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.

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.
- Use `inc` and `dec` to increment or decrement variables.
- Each time `Sync()` is executed a new screen frame is created.
- The current screen frame is held in the front buffer, the next frame in the back buffer.
- Use `Timer()`, `ResetTimer()`, `GetSeconds()` and `GetMilliSeconds()` to access how long a program has been running.
- Use `Sleep()` to get a program to pause for a specified number of milliseconds.
- Use `RandomSeed()`, `Random()` and `RandomSign()` to generate random integer values.

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 ***
sqrt# = no^0.5
rem *** Display result ***
PrintC("Square root of ")
PrintC(no)
PrintC(" is ")
Print(sqrt#)
Sync()
do
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

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 = $12 / 6 * 5 - 25$
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"  
Print("Hello, ")  
Print(name$)  
Print(", how are you today?")  
Sync()  
do  
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"  
Print("Hello, "+name$+", how are you today?")  
Sync()  
do  
loop
```

Activity 3.16

Modified code:

```
do  
rem *** Get time passed ***  
time_elapsed# = Timer()  
rem *** Display time ***  
Print("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(":")
    Print(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(":")
    Print(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)
rem *** Display value thrown ***
PrintC("Value thrown was : ")
Print(dice)
Sync()
do
loop
```

Activity 3.23

The colours change so quickly that there is no 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)
rem *** Display value thrown ***
PrintC("Value thrown was : ")
Print(dice)
Sync()
do
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()
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
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.

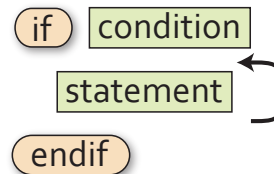
if

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

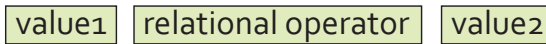
```
no < 0
```

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`

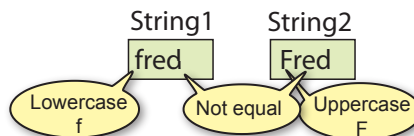
When two strings are checked for equality as in the condition

```
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
```

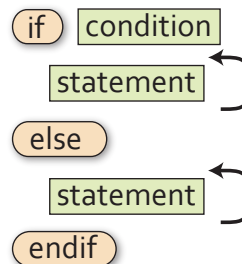
Although this will work, it's not very efficient since we always have to test both conditions - and the second condition can't be true if the first one is! As an alternative, we can add the word `else` to our original `if` statement and follow this by the action we wish to have carried out when the stated 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:

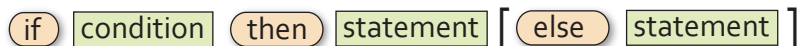
```
Display a prompt message for first number
Read the first number
Display a prompt message for the second number
Read the second number
IF first number is less than the second number THEN
    Set answer to first number
ELSE
    Set answer to second number
ENDIF
Display answer
IF answer is an even number 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

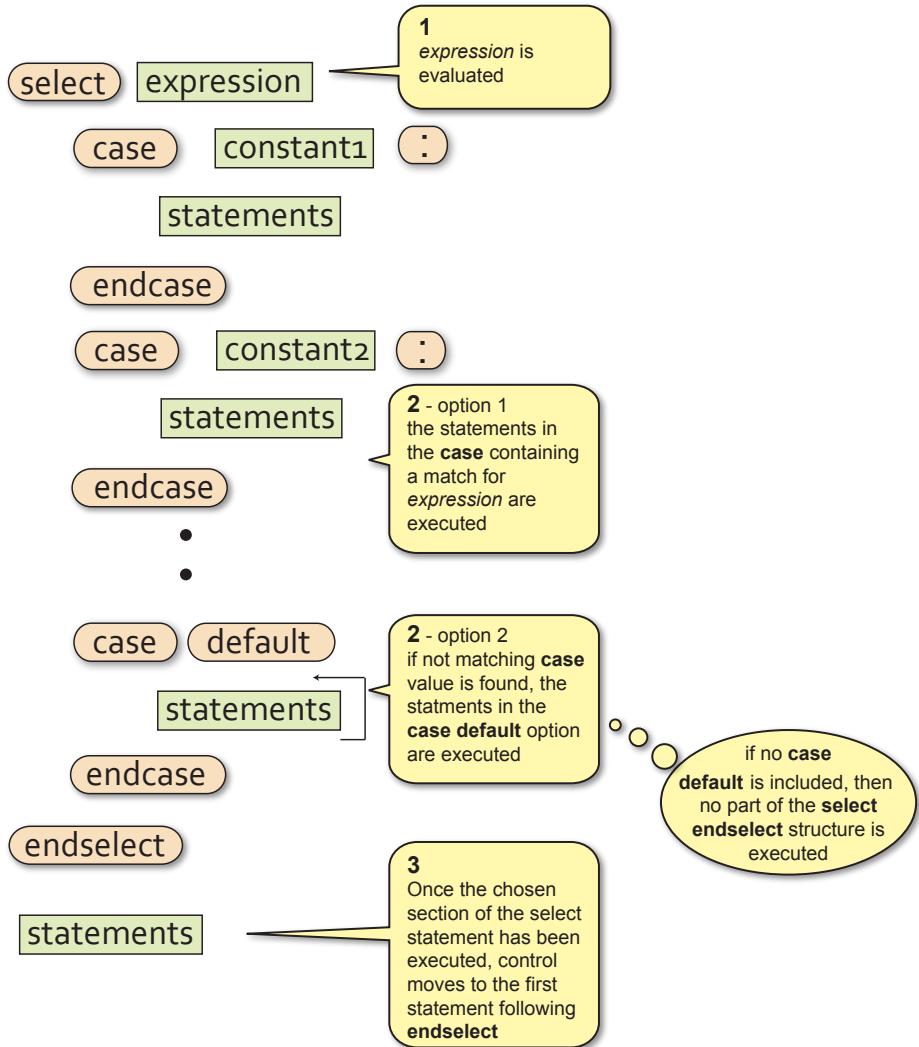
```
  case 7:
    Print("Saturday")
  endcase
  case default
    Print("Invalid day")
  endcase
endselect
```

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
endsselect
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:

GetName() is assumed to be a user-written function that allows the player to enter their name.

```
name$ = GetName()
select name$
  case "Jack","Jill" :
    Print("Hello friend")
  endcase
  case default
    Print("I do not know your name")
  endcase
endselect
Sync()
```

Although the `case` value may also be a real value as in the line

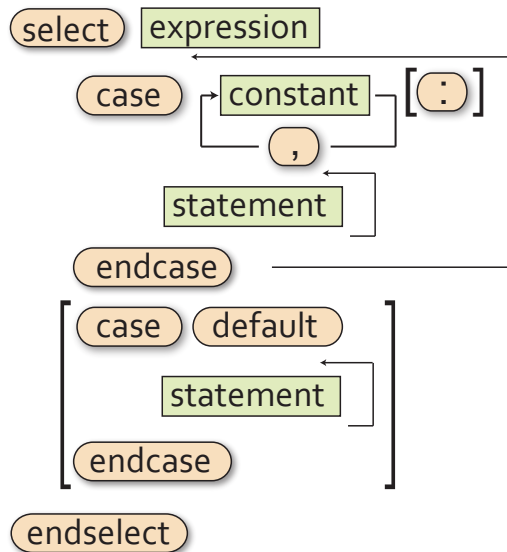
```
CASE 1.52
```

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

- Valid.
- Valid.
- Valid.
- Invalid. \Rightarrow is not a relational operator (should be \geq).
- Invalid. Integer variable compared with string.
- Invalid. 14 High Street should be in double quotes.

Activity 4.2

- False. Only the second string contains a space.
- True. "def" is shorter and matches the first three characters of "defg".
- True. "A" comes before "B".
- False. Only the second string contains a full stop.
- False. Only the second string contains a capital D.
- 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:

```
while floor_area = 25
  health = health - 10
endwhile
```

The code assumes a variable called *floor_area* records the position of the character and that the "bad health" area is at position 25.

The syntax of AGK BASIC's **while .. endwhile** construct is shown in FIG-5.1.

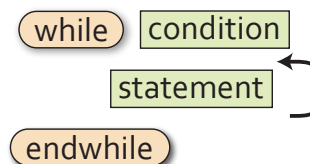


FIG-5.1

while..endwhile

AGK BASIC's **while** statement does not use the term **do**.

where:

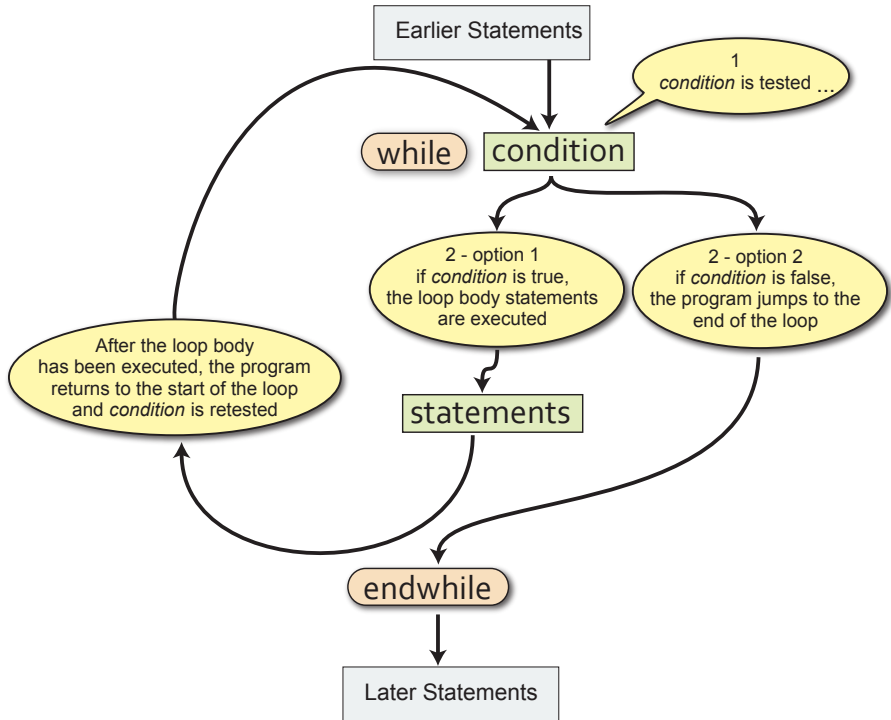
- | | |
|------------------|--|
| condition | is a Boolean expression and may include and , or , not and parentheses as required. |
| statement | is any valid AGK BASIC statement. |

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

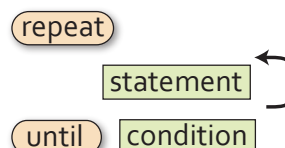
```
total = 0
repeat
    number = GetButtonEntry()
    total = total + number
until number = 0
```

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:

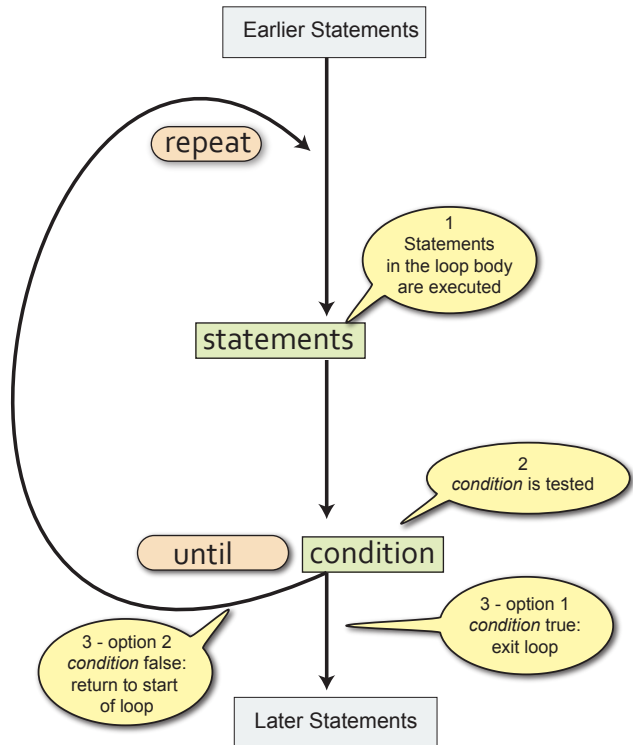
condition is a Boolean expression and may include **and**, **or**, **not** and parentheses as required.

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 1×7 to 12×7 .

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 1.1000002384.

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

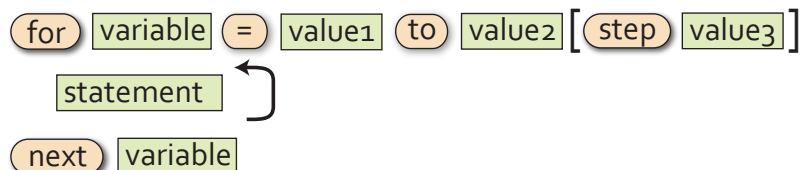
Modify *ForReal* so that the upper limit of the loop is 2.01.

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.

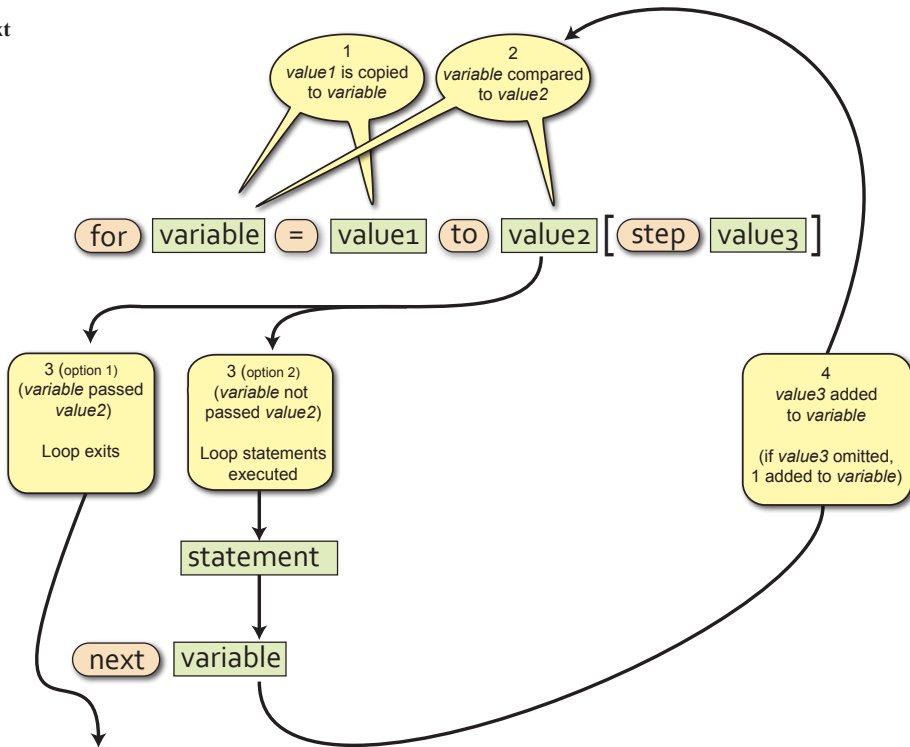
value3 is the value to be added to the loop counter after each iteration. If this is omitted then a value of 1 is added to the loop counter.

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.

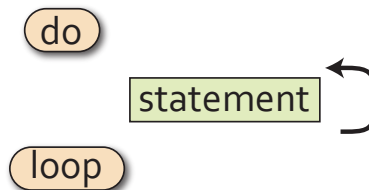
When a `do` loop is executing, then, under normal circumstances, the program will only terminate when forced to do so by an external event. In all our projects so far, the external event has been the operating system closing down our program in response to our clicking on the X button at the top-right 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 AKG 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

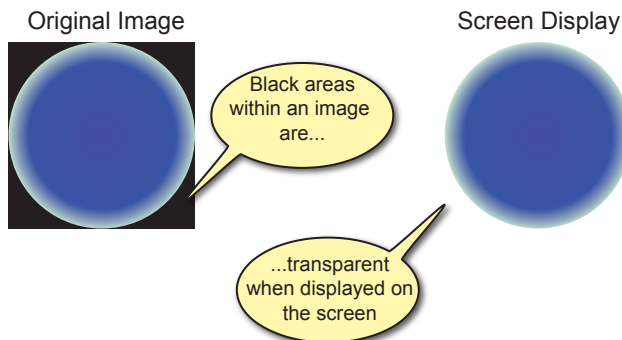
is a lossy format and is ideal for use with photographs and larger graphics. The degree of compression used when saving a file in JPG format can be specified. Less compression means a better quality image but a larger file.

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



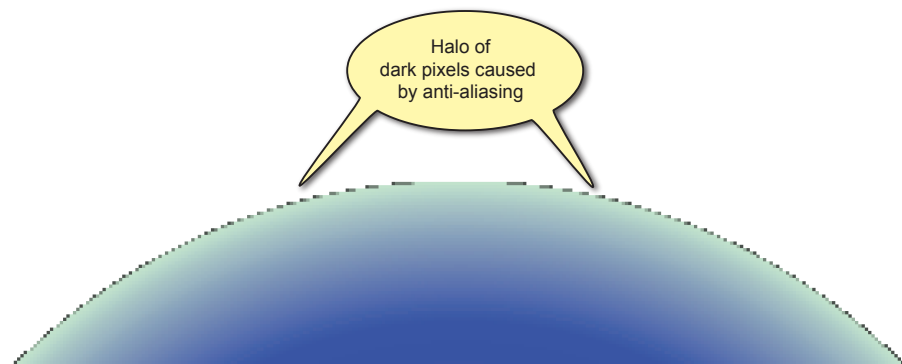
However, there are three things to be careful of when using this option:

- 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.

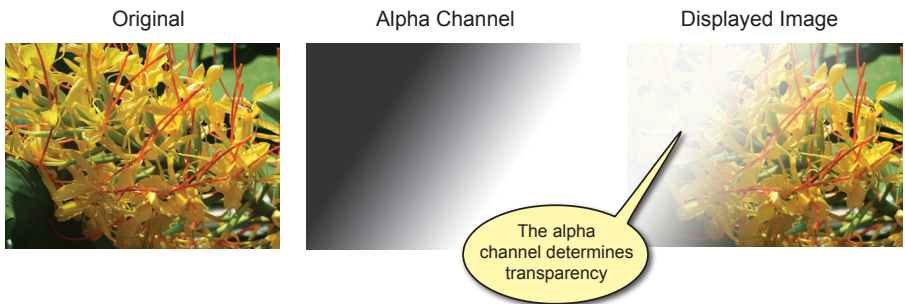
A second option for creating transparency is to include an **alpha channel** in the image itself.

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

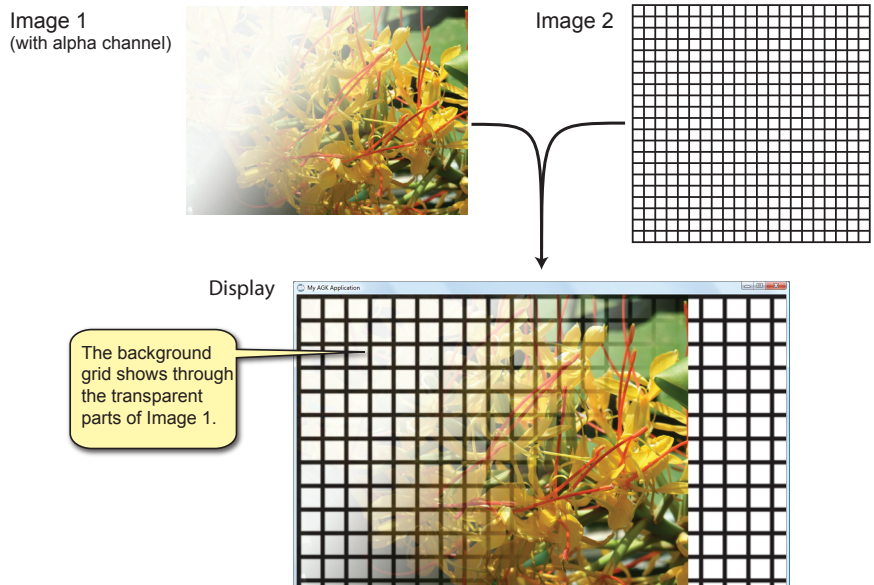
An Image with an Alpha Channel



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 *media* 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 *iflag* 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.

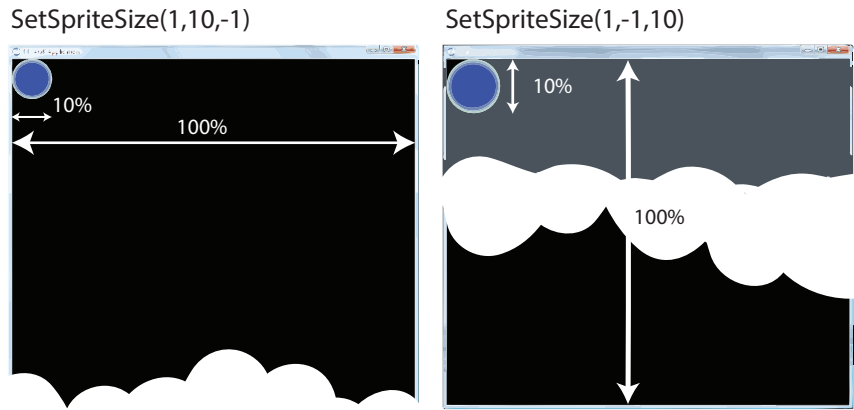
Of course, setting the *fy* to 10 and *fx* to -1 with

```
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



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.

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-6.17).

FIG-6.17

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.

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-6.18).

FIG-6.18

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.

Automatically assigned ID values start at 1.

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-6.19).

FIG-6.19

PlaySound()

`PlaySound (id [, ivol [, iloop [, iprty]]])`

where:

id is an integer value specifying the ID previously assigned to the sound.

Several sound files can be played at the same time.

- 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.

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-6.20.

FIG-6.20

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-6.21 for format).

FIG-6.21

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-6.22.

FIG-6.22

SetSoundSystemVolume()

`SetSoundSystemVolume (ivol)`

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-6.23).

FIG-6.23

GetSoundExists()

integer `GetSoundExists (id)`

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-6.24).

FIG-6.24

GetSoundsPlaying()

integer `GetSoundsPlaying` ((`id`))

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-6.25.

FIG-6.25

GetSoundInstances()

integer `GetSoundInstances` ((`id`))

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.

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-6.26.

FIG-6.26

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-6.27).

FIG-6.27

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.

idFin is an integer value giving the highest ID of the list of music files to be played.

Only one music file can be playing at any one time.

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

These tracks would have to be stored in the project's *media* folder.

```
LoadMusic(1,"TrackA.mp3")
LoadMusic(2,"TrackB.mp3")
LoadMusic(3,"TrackC.mp3")
LoadMusic(4,"TrackD.mp3")
LoadMusic(5,"TrackE.mp3")
```

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-6.28.

FIG-6.28

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-6.29).

FIG-6.29

ResumeMusic()

```
ResumeMusic ( )
```

StopMusic()

To stop a music file completely use `StopMusic()` (see FIG-6.30).

FIG-6.30

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-6.31).

FIG-6.31

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-6.32).

FIG-6.32

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-6.33).

FIG-6.33

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-6.34).

FIG-6.34

SetMusicSystemVolume ((ivol))

SetMusicSystemVolume()

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-6.35.

FIG-6.35

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-6.36.

FIG-6.36

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-6.37.

FIG-6.37

GetPointerState()

integer GetPointerState (())

The code in FIG-6.38 demonstrates the use of the `GetPointerPressed()` and `GetPointerReleased()` statements.

FIG-6.38

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-6.38.

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-6.39.

FIG-6.39

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-6.40.

FIG-6.40

integer `GetSpriteHit` ((fx , fy))

`GetSpriteHit()`

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.

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-6.41.

► Text resources use the same character images as `Print()` to form the displayed text.

FIG-6.41

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-6.42

We can select the color and transparency of the text using the `SetTextColor()` statement (see FIG-6.42).

SetTextColor()

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.

itrans is an integer value specifying the opaqueness of the text. Range 0 (invisible) to 255 (fully opaque).

The default colour for a text resource is white.

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-6.43).

FIG-6.43

`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.

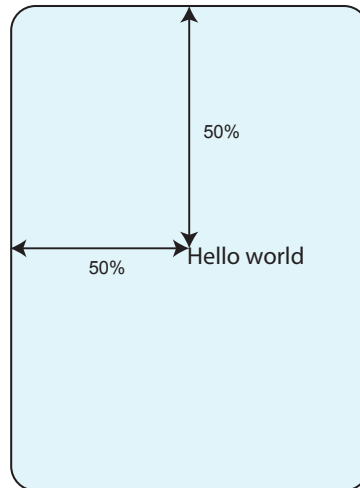
We could place text resource 1 at the centre of the app window using the statement:

```
SetTextPosition(1,50,50)
```

The position (50,50) refers to the top left part of the text (see FIG-6.44).

FIG-6.44

Positioning a Text Resource



SetTextSize()

The size of the text can be adjusted using the `SetTextSize()` statement (see FIG-6.45).

FIG-6.45

`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-6.46).

FIG-6.46

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-6.47).

FIG-6.47

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-6.48).

FIG-6.48

DeleteText()

```
DeleteText ( ( id )
```

where:

id is an integer value giving the ID of the text resource to be deleted from the program.

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-6.49.

FIG-6.49

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
    rem *** Empty text resource ***
    SetTextString(1,"")
    Sync()
    rem *** Wait one second ***
    Sleep(1000)
next c
rem *** Delete resource ***
DeleteText(1)
do
loop
```

Activity 6.19

Start a new project called *UsingText* and modify the code in *main.agc* to match that given in FIG-6.49. 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)
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 : *AGKSplash.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

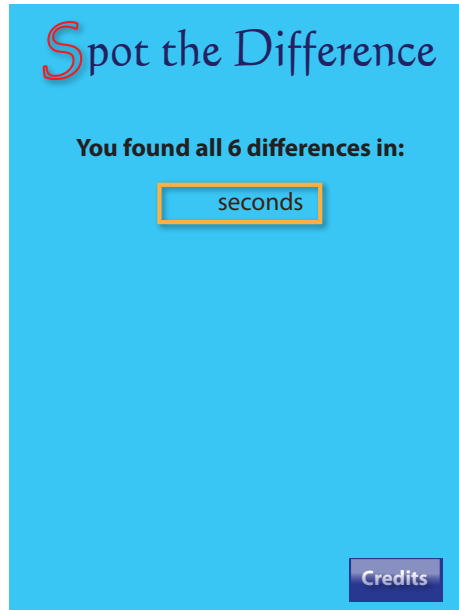


The photos themselves are not separate entities but part of the single overall image. Note that the top right corner leaves a gap where the time is to be displayed in real-time.

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

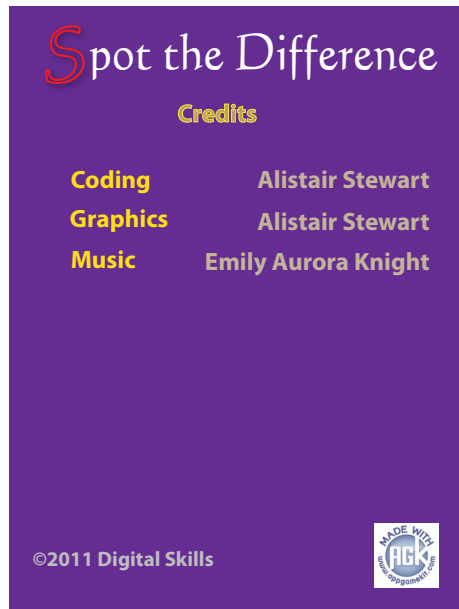


Again, you can see that a space has been left for the actual number of seconds taken to find all the differences. In addition, this screen also shows a separate button sprite in the bottom-right which allows the user to view the credits screen if required.

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



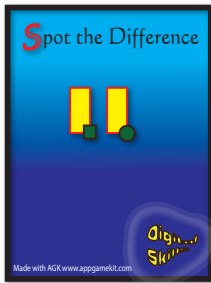
Other Resources

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

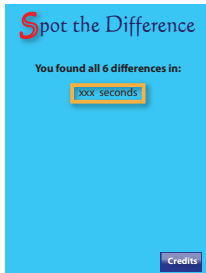


Splash Screen



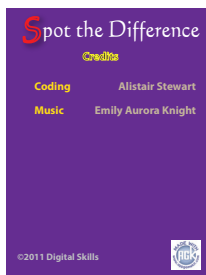
Main Screen

- Music begins
- Rings appear around correctly selected areas
- Sound effect when ring appears
- Time in seconds displayed



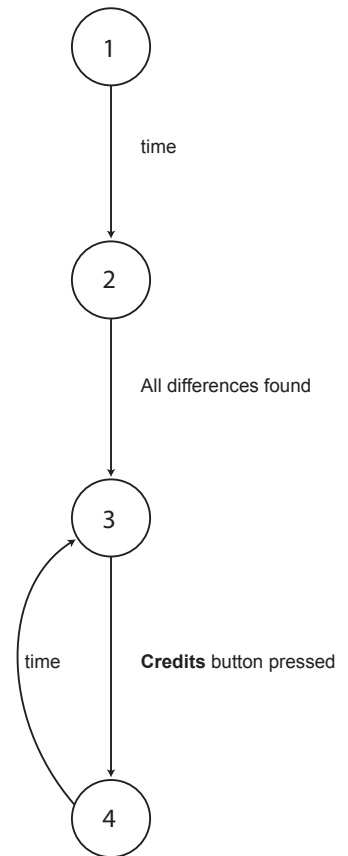
End Screen

- 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.

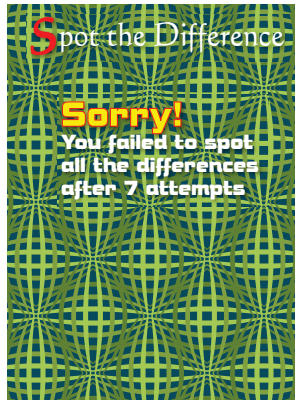
Test this modification and check that the player has time to see the final ring in position before the *End* screen appears.

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



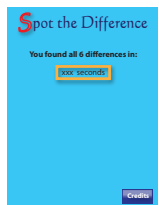
Main Screen

- Music begins
- Rings appear around correctly selected areas
- Sound effect when ring appears
- Time in seconds displayed



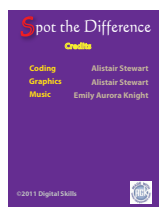
Failed Screen

- Music continues
- Displays failed message



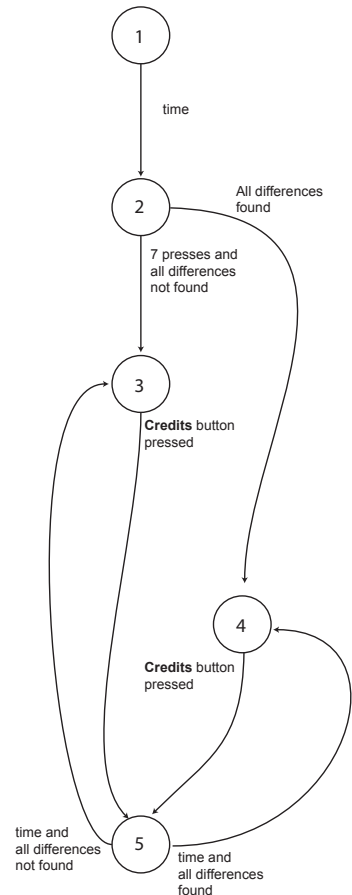
End Screen

- 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
```

You have reached the end of this extract from

Hands On AGK BASIC

by Alistair Stewart

You can purchase the complete publication (approx 900 pages) in either printed or ebook format from

The Games Creators [printed version only]

<http://www.thegamecreators.com/>

or

Digital Skills [printed or ebook (PDF)]

www.digital-skills.co.uk

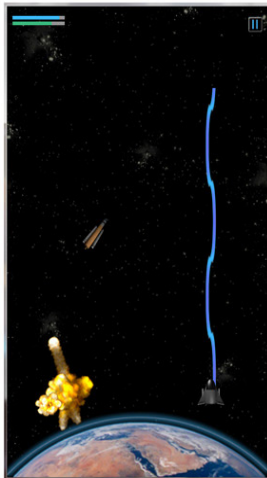
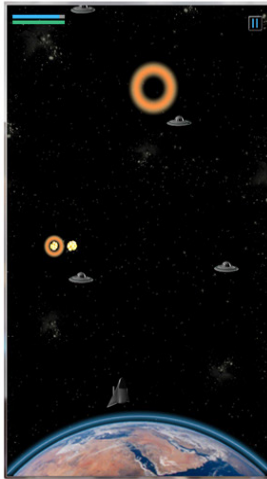
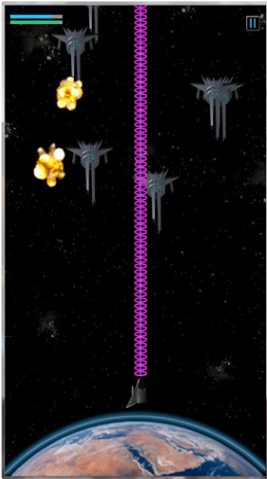
Other books by Alistair Stewart include:

Hands On DarkBASIC Pro Volume 1

Hands On DarkBASIC Pro Volume 2

Hands On Milkshape

DFENZ



Visit NaplandGames.com for more info