Alistair Stewart

The Official
App**Game****Kit**
Tutorial Guide

Digital Skills

the
GameCreators

The Official
App Game Kit Tutorial Guide

Alistair Stewart

Digital Skills
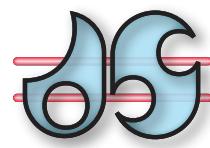
Milton
Barr
Ayrshire
KA26 9TY
UK

www.Digital-Skills.co.uk

Contents

Data

Iteration and Debugging

A First Look at Resources

Spot the Difference Game

User-Defined Functions

String and Maths Functions

Arrays

Records and Bit Operators

File Handling

Multimedia Resources

Sprite Animation

Creating Games

Sync() and Images

Text Resources

Particles

Advanced Sprites

Screen Handling

User Input - 2

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. We 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 should be clear and unambiguous

- Each instruction should be as brief as possible

Solving Problems

As programmers, we will normally be presented with a problem for which we are expected to come up with an efficient algorithm and computer program.

The first step is to make sure that the problem to be tackled has been stated in a clear and unambiguous way. If it isn't, ask for a clearer, more detailed description of what is required.

The second step is to make sure you understand the problem. If you don't, try reading the problem again or getting help from others.

The third step is to develop some method of tackling the problem. Remember, the first ideas you come up with are often not the best ones!

The final step is to create a solution to the problem - the algorithm.

Let's take a favourite logic problem and see how we can develop a solution:
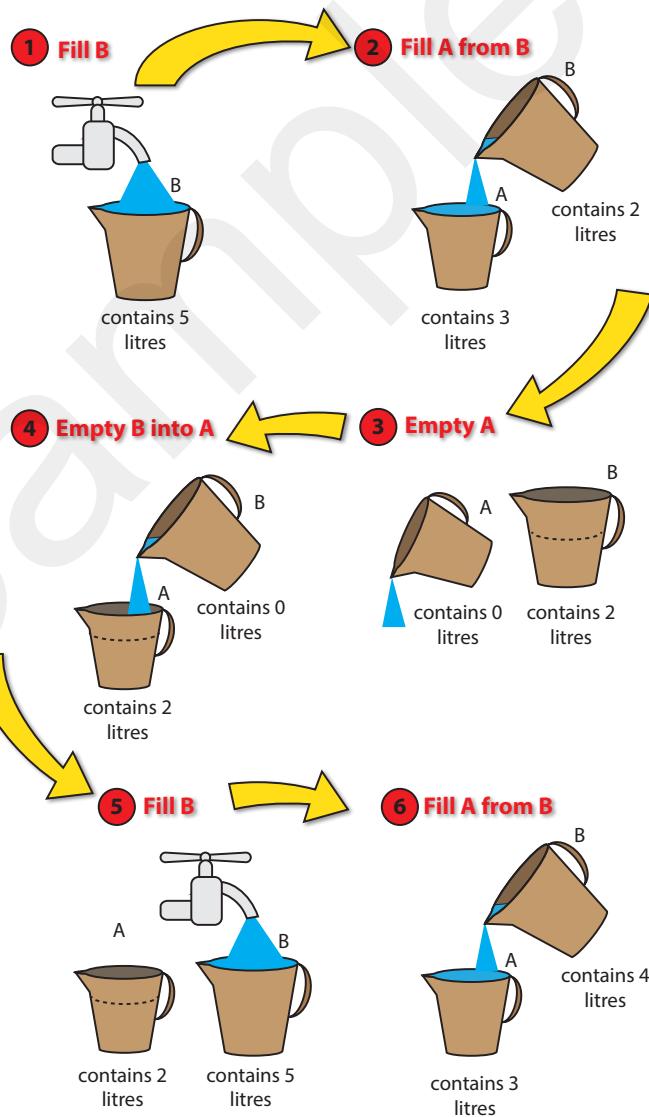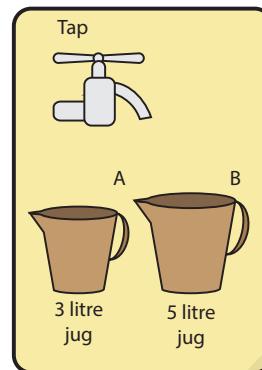
A container (labelled A) holds exactly 3 litres of water when full. A second container

(labelled B) holds exactly 5 litres. With an unlimited water supply, measure out exactly 4 litres of water.

Assuming we understand the problem, FIG-1.1 shows how we might go about tackling the problem.

FIG-1.1

Solving the Four Litre Problem

The solution to the task can now be written as the following algorithm:

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

Activity 1.2

Often there will be more than one way to arrive at a solution to a problem. Write an alternative solution to the 4 litre problem by starting with the instruction

```
Fill A
```

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

Activity 1.3

Write an algorithm for the following problem:

A traveller arrives at a river he must cross with a wolf, goat and cabbage. He has access to a boat to cross the river and reach home. However, the boat can only carry him and one of his belongings. Normally, he would just make several crossings to get the wolf, goat and cabbage to the other side, but, if left alone together, the wolf would eat the goat and the goat would eat the cabbage.

How does the traveller get all three possessions safely to the other side of the river?

Computer Programs

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. For example, the code below displays the result of the calculation 12 x 3.

```
num1 = 12
num2 = 3
answer = num1 * num2
Print(answer)
```

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.

Computer programs are normally **copied** (or **loaded**) from a magnetic disk or flash storage 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 300,000 million (or 300 billion) instructions per second (usually written as 300,000 **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 game-oriented, an area of programming for which AGK BASIC has been specifically designed.

The Nature of Algorithms

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 for us to move from the rather free-style way we might express these structures in everyday English to the more formalised style used for writing algorithms. This formalisation better matches the structures used within 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, the solutions to the 4 litre problem and traveller problems were both examples of a sequence.

> **Activity 1.4**
>
> Download the file containing support material for this book from *www.digital-skills.co.uk* (you'll find a link on the **AGK** *Downloads* page) and unzip the file.
>
> From the folder *AGK2/Resources/Ch01/TriLogic,* run *TriLogicGame.exe,* press the **Start** button. Click on the tokens to construct the lines of an algorithm that moves the counters from position 1 to position 3.
> A typical statement would be
> ```
> MOVE C1 TO P2
> ```
>
> Note that a larger piece must never be placed on top of a smaller piece.

As you can see from the *TriLogic* puzzle, you are expected to construct the instructions in a very specific format. If you deviate from that format, you'll get an error message. This is typical of any true programming language: each statement must be constructed according to strict rules.
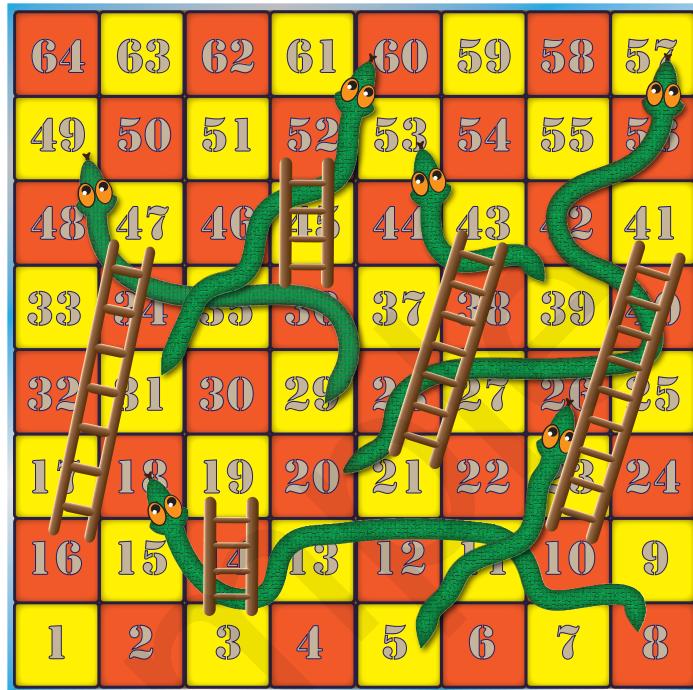
Selection

Binary Selection

Often a group of instructions in an algorithm should be carried out only when certain circumstances arise. For an example of this, consider the board game of Snakes and Ladders (see FIG-1.2).
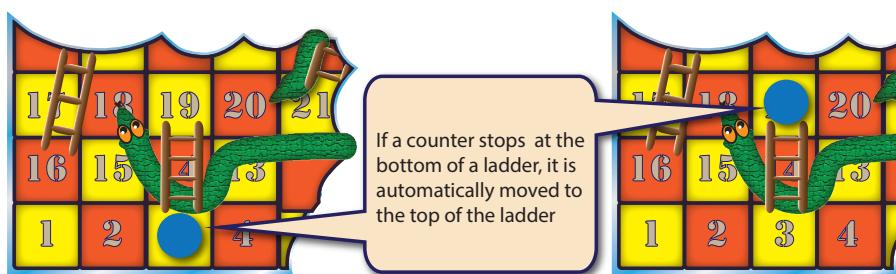
Each player has a counter which moves along the board by an amount determined by the throw of a die. The aim of the game is to be the first to reach the final square (square 64). We could describe a move as

```
Throw die
Move counter forward by the number thrown
Pass die to next player
```

However, when a player's counter stops on a square at the bottom of a ladder, it moves to the square at the top of the ladder (see FIG-1.3).

If a counter stops at the bottom of a ladder, it is automatically moved to the top of the ladder

We might explain this rule with an instruction such as:

> When a counter stops at the bottom of a ladder, move the counter to the top of the ladder

Notice that the statement consists of two main components:
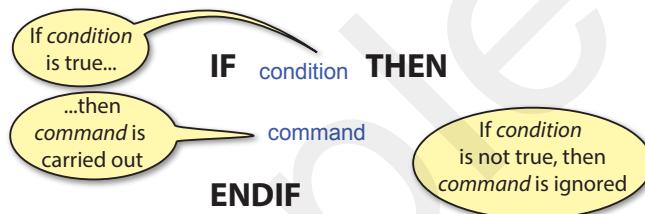
     a condition   : a counter stops at the bottom of a ladder

and

     a command   : move the counter to the top of the ladder

A **condition** (also known as a **Boolean expression**) is a statement that is either true or false at a given moment in time. 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**.

Although we could rewrite the above instruction in many different ways, when we produce a set of instructions in a formal manner, as we are required to do when writing algorithms, then we use a specific layout as shown in FIG-1.4, always beginning with the word IF.

It is important to realise that there are two alternative options to this structure: to carry out the command or to ignore it. From this we get the formal name for the IF statement - **binary selection**.

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 the instruction. Appropriate indentation is particularly valuable in aiding readability once an algorithm becomes long and complex. Using this layout, the instruction for our Snakes and Ladders game would be written as:

```
IF counter stops at the bottom of a ladder THEN
    Move counter to top of ladder
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, we could now write the instruction for a single move in Snakes and Ladders as:

```
Roll die
Move counter forward by the number thrown
IF counter stops at the bottom of a ladder THEN
      Move counter to top of ladder
ENDIF
Pass die to next player
```

This longer list of instructions highlights the usefulness of the term ENDIF in separating the conditional command, Move counter to top of ladder, from subsequent unconditional instructions, in this case, Pass die to next player.

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 he thought of.

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

An algorithm may contain many separate IF statements if the logic requires them. For example, in Snakes and Ladders, another rule is that counters that stop at the head of a snake must move to the tail of the snake.
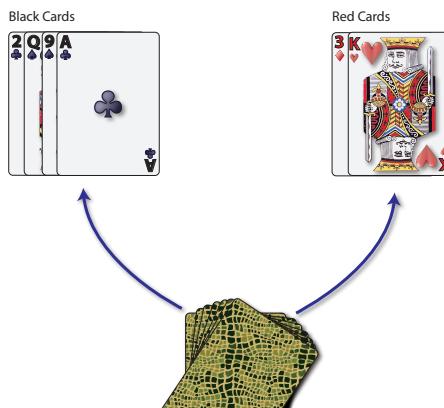
Activity 1.6

Modify the algorithm of Snakes and Ladders, given earlier, to include instructions for landing on the head of a snake.

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, lets assume that a card game requires the top card of a face-down deck is to be turned face up and then added to the left or right hand pile as appropriate (see FIG-1.5).

FIG-1.5

Placing Cards

Black Cards   Red Cards
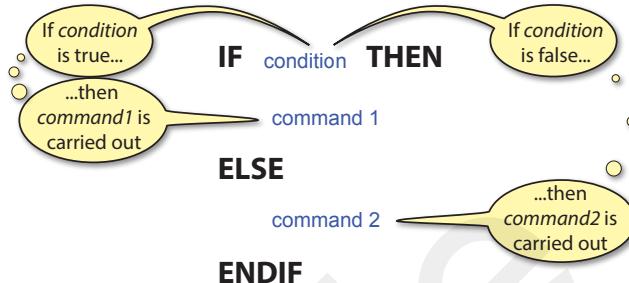
We could write the algorithm for this as:

```
Turn over top card
IF card is black THEN
        Add card to left-hand pile
ELSE
        Add card to right-hand pile
ENDIF
```

The general form of this extended IF statement is shown in FIG-1.6.
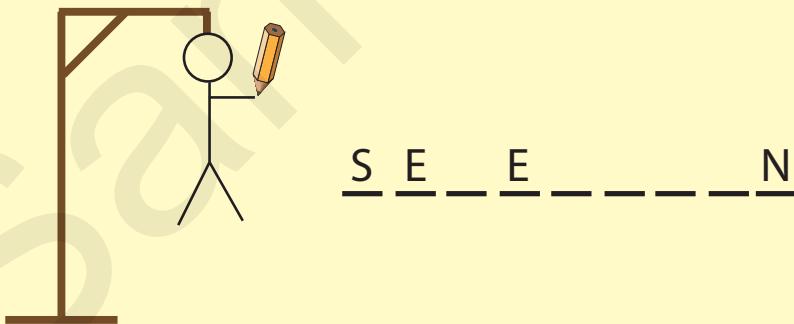
FIG-1.6

The IF..THEN..ELSE
Structure

Activity 1.7

In the game of Hangman, *Player 1* has to guess the letters in a word known to *Player 2*. At the start of the game *Player 2* draws one underscore for each letter in the word. When *Player 1* guesses a letter which is in the word, *Player 2* writes the letter above the appropriate underscore. 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
```

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 guess matches number THEN
     Player 1 says "Correct"
ENDIF
IF guess is lower than number THEN
     Player 1 says "Too low"
ENDIF
IF guess is higher than number THEN
     Player 1 says "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 (see FIG-1.7).

FIG-1.7

Working Through the Algorithm

Lets assume we want to work through the algorithm below when the guess is 4 and the number is also 4.	In this situation, the first condition is true, and so we execute the conditional statement in the first IF structure.

IF guess matches number THEN
          Player 1 says "Correct"
ENDIF
IF guess is lower than number THEN
          Player 1 says "Too low"
ENDIF
IF guess is higher than number THEN
          Player 1 says "Too high"
ENDIF
 	
true
	IF guess matches number THEN
	          Player 1 says "Correct"
	ENDIF
	IF guess is lower than number THEN
	          Player 1 says "Too low"
	ENDIF
	IF guess is higher than number THEN
	          Player 1 says "Too high"
	ENDIF
	
	executed

After completing the first IF statement, the conditions in the second and third IF statements are tested and found to be false.	If we examine the three conditions more closely, we can see that only one of them can be true at any given moment.

IF guess matches number THEN
          Player 1 says "Correct"
ENDIF
IF guess is lower than number THEN
          Player 1 says "Too low"
ENDIF
IF guess is higher than number THEN
          Player 1 says "Too high"
ENDIF
 	
false   *false*
	guess matches number
	
	guess is lower than number
	
	guess is higher than number
	
	Only one condition can be true for a given set of values

Checking those last two IF statements is a waste of effort 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 false, do we bother to examine the other conditions being tested. So, for example, in Snakes and Ladders, we cannot be at the bottom of a ladder and at the head of a snake at the same time, so we could rewrite our IF statements as

```
IF counter at bottom of ladder THEN
    Move counter to top of ladder
ELSE
    IF counter at head of snake THEN
        Move counter to tail of snake
    ENDIF
ENDIF
```

In the number guessing game, we have three possible outcomes to handle. Taking things slowly, we could start this part of our algorithm with:

```
IF guess matches number THEN
    Player 1 says "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). But what are these other conditions suggested by this statement? They are

```
guess is lower than number
```
and
```
guess is higher than number
```

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 lower than number THEN
    Player 1 says "Too low"
ELSE
    Player 1 says "Too high"
ENDIF
```

Now, by replacing the phrase ***Check the other conditions*** in our original algorithm with our new IF statement we get:
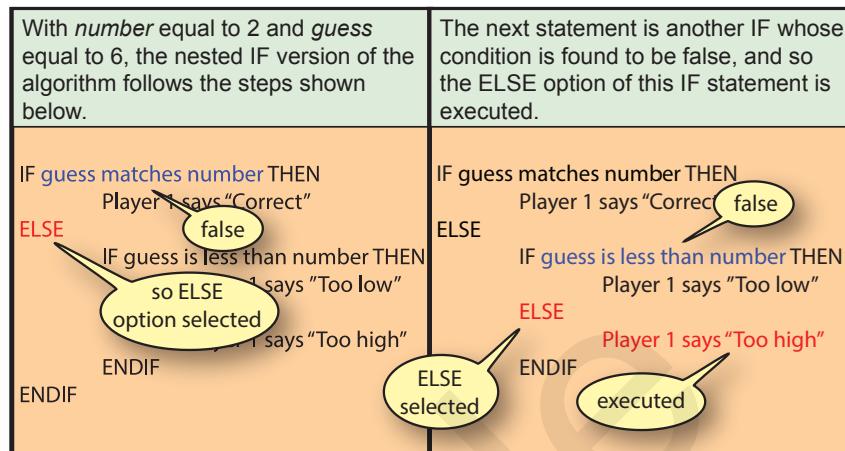
```
IF guess matches number THEN
    Player 1 says "Correct"
ELSE
    IF guess is less than number THEN
        Player 1 says "Too low"
    ELSE
        Player 1 says "Too high"
    ENDIF
ENDIF
```

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**.

FIG-1.8 shows how our new nested IF algorithm handles the situation where *number* is 2 and *guess* is 6.

FIG-1.8

Working Through A
Nested IF Structure

With *number* equal to 2 and *guess* equal to 6, the nested IF version of the algorithm follows the steps shown below.	The next statement is another IF whose condition is found to be false, and so the ELSE option of this IF statement is executed.

```
IF guess matches number THEN
        Player 1 says "Correct"        false
ELSE
        IF guess is less than number THEN
                ...1 says "Too low"     so ELSE
                                        option selected
                ...1 says "Too high"
        ENDIF                           ELSE
ENDIF                                   selected
```

```
IF guess matches number THEN
        Player 1 says "Correct"        false
ELSE
        IF guess is less than number THEN
                Player 1 says "Too low"
        ELSE
                Player 1 says "Too high"
        ENDIF                           executed
```

Activity 1.9

Show what parts of the nested IF algorithm would be performed if *number* was 8 and *guess* 7.

Where there are even more mutually exclusive alternatives, several IF statements may be nested in this way.

Activity 1.10

In a video game, the player controls the movement of a character using the following keys:

Key	Movement
W	Forward
S	Backward
M	Move to the right
N	Move to the left
U	Jump up

The character fires when the space key is pressed.

Only one key can be pressed at a time.

Write a set of nested IF statements which can handle the situation described above.

The solution should contain lines such as

```
        IF W key pressed THEN
and
        Move character forward
```

As you can see from the solution to Activity 1.10, 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 1.10 as:

```
IF
    W key pressed:        Move character forward
    S key pressed:        Move character backward
    M key pressed:        Move character to the right
    N key pressed:        Move character to the left
    U key pressed:        Make character jump up
    Space key pressed:    Fire weapon
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 six options; there can be as many as the situation requires.

We could add another feature to our character controls by making the game emit a sound when an invalid key (such as X or B) is pressed. To do this we would add an extra ELSE section to our code.
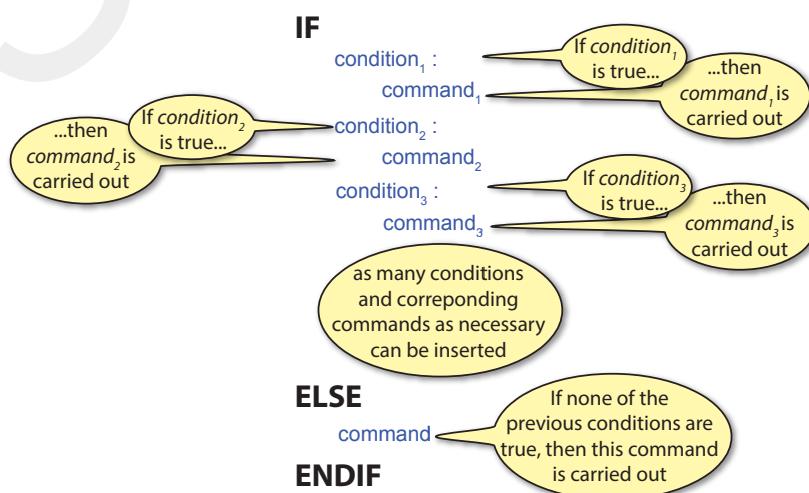
```
IF
    W key pressed:        Move character forward
    S key pressed:        Move character backward
    M key pressed:        Move character to the right
    N key pressed:        Move character to the left
    U key pressed:        Make character jump up
    Space key pressed:    Fire weapon
ELSE
    Play beep noise
ENDIF
```

The additional ELSE option will be chosen only if none of the other options are applicable (that is, if an invalid key is pressed). 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-1.9.

FIG-1.9

The Multi-Way IF Structure


```
IF
    condition₁ :
        command₁
    condition₂ :
        command₂
    condition₃ :
        command₃
ELSE
    command
ENDIF
```

If condition₁ is true... ...then command₁ is carried out

...then command₂ is carried out   If condition₂ is true...

If condition₃ is true...   ...then command₃ is carried out

as many conditions and correponding commands as necessary can be inserted

If none of the previous conditions are true, then this command is carried out

Iteration

There are certain circumstances in which it is necessary to perform the same sequence of instructions several times. For example, during a lottery draw, we could describe the basic action as

    Pick out ball
    Call out number on the ball

Now, since six balls are drawn, we need to perform these instructions six times. One way to create an algorithm for this task is simply to repeat the statements:

    Pick out ball
    Call out number on the ball
    Pick out ball
    Call out number on the ball
    Pick out ball
    Call out number on the ball
    Pick out ball
    Call out number on the ball
    Pick out ball
    Call out number on the ball
    Pick out ball
    Call out number on the ball

This would certainly accomplish the task, but it is rather-long winded.

However, not only does it seem rather time-consuming to have to write the same sequence of instructions six times, but it would be even worse if we used the same approach to describe a game of Bingo where many more balls are drawn!

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 that we want to perform over and over again are known as the **loop body**.

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 lottery problem our statement would be:

    FOR 6 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 6 times DO
        Pick out ball
        Call out number on ball

Finally, to mark the fact that we have reached the end of the loop body statements we add the word ENDFOR:

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

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

Activity 1.13

If we were required to draw out 10 balls rather than 6, what changes would we need to make to the algorithm?

The latest algorithm for our guessing game was:

    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 1.14

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
```

> **Activity 1.15**
>
> Can you see a practical problem with the algorithm?
>
> If not, try playing the game a few times, playing exactly according to the instructions in the algorithm.

Occasionally, we may have to use a slightly different version of the FOR loop.

In our *Snakes and Ladders* game we described a player's move with the algorithm

```
Roll die
Move counter forward by the number thrown
IF counter stops at the bottom of a ladder THEN
    Move counter to top of ladder
ENDIF
IF counter stops at the head of a snake THEN
    Move counter to tail of snake
ENDIF
Pass die to next player
```

Although we need to have every player perform these same instructions, we have no way of knowing, when writing the instructions, exactly how many players there will be each time the game is played. To overcome this problem we start our loop with the statement

```
FOR each player DO
```

to give the following algorithm

```
FOR each player DO
    Roll die
    Move counter forward by the number thrown
    IF counter stops at the bottom of a ladder THEN
        Move counter to top of ladder
    ENDIF
    IF counter stops at the head of a snake THEN
        Move counter to tail of snake
    ENDIF
    Pass die to next player
ENDFOR
```

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
```
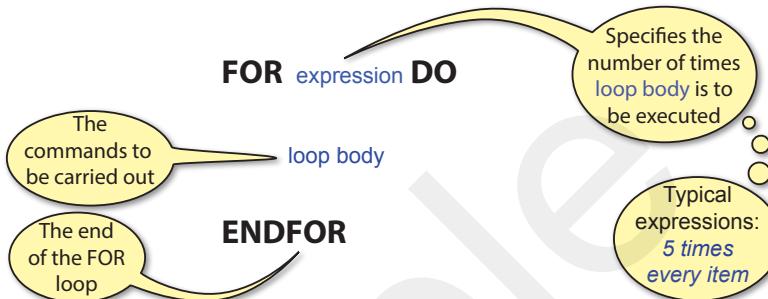
> **Activity 1.16**
>
> A card game requires only cards showing values between 1 (Ace) and 7, so before beginning the game, all other cards (8 to King) must be removed.
>
> Write an algorithm which places all cards in the range 1 to 7 in a separate pile from those cards showing other values.

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

FIG-1.10

The FOR..ENDFOR Loop

FOR expression **DO**

Specifies the number of times loop body is to be executed

The commands to be carried out → loop body

Typical expressions: *5 times every item*

The end of the FOR loop → **ENDFOR**

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, the algorithm created for the guessing game in Activity 1.15 highlighted the problem of having a fixed number of attempts at guessing the value of a number. 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 term. The UNTIL term 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:
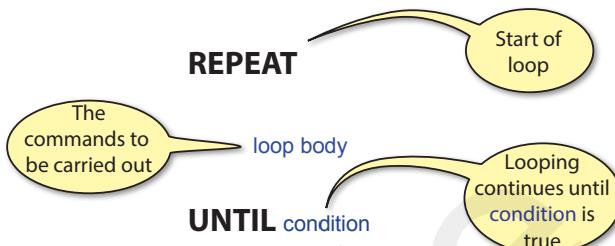
```
REPEAT
    Put coin in machine
    Pull handle
    IF you win THEN
        Collect winnings
    ENDIF
UNTIL you want to stop
```

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

FIG-1.11

The REPEAT..UNTIL
Loop

REPEAT

> Start of loop

> The commands to be carried out

loop body

> Looping continues until condition is true

UNTIL condition

Activity 1.17

A game requires a player to make use of a shuffled pack of cards lying face-down. The top card is turned over and discarded. This continues until an Ace is turned over.

Using REPEAT..UNTIL, write the logic required for the game.

WHILE.. ENDWHILE

A final method of iteration, differing only subtly from the REPEAT.. UNTIL loop, is the WHILE .. ENDWHILE structure which causes the statements in the loop body to be executed as long as the stated condition is true. The condition appears at the start of the structure (beside the term WHILE) and is known as an **entry condition**. 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 take another card while his cards have a total value of less than 17. So we might attempt to write the rules for the dealer as:

```
Calculate the value of the initial two cards in hand
REPEAT
    Take another card
UNTIL value of cards in hand 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 17 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 rewrite the logic as
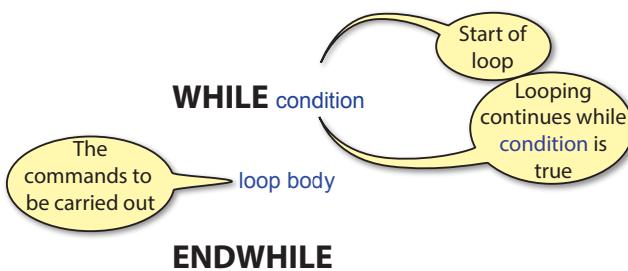
```
Calculate the value of the initial two cards in hand
WHILE value of cards in hand is less than 17 DO
    Take another card
ENDWHILE
```

Now determining if the value 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

The differences in operation between the REPEAT..UNTIL and the WHILE.. ENDWHILE structures are shown in FIG-1.13.

FIG-1.13

The Differences
Between REPEAT..
UNTIL and WHILE..
ENDWHILE.

REPEAT..UNTIL	WHILE..ENDWHILE
The condition appears after the loop body	The condition appears before the loop body
Looping stops when the condition becomes true	Looping stops when the condition becomes false

The main consequence of these differences 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 1.18

A game involves throwing two dice. If the two values thrown are not the same, then the die 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

                Roll both dice
and   Choose die with lower value

Complex Conditions

We have encountered the use of conditions in the IF, REPEAT..UNTIL, and WHILE.. ENDWHILE structures but so far we have shown only simple conditions in the examples given. More complex conditions can be specified using the same terms we might employ in everyday conversation: AND, OR and NOT.

The AND Operator

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
```

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 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:

```
Determine the truth of condition 1
Determine the truth of condition 2
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 condition 1 (*it's dark*) is true but condition 2 (*motion has been detected*) is false. Because one of the conditions is false, the overall result is false and the light does not come on.

You are not limited to just one AND operator in a complex condition; you can have as many as you need. For example, the conditions for foreign national flying to the USA can be written as

```
IF you have a passport AND you have a visa AND you have an airline ticket THEN
    You can fly to the USA
ENDIF
```

All three conditions must be true before you can fly to the USA.

Activity 1.19

A person must meet the following conditions to apply for a job:
    age over 21 AND height at least 5 feet 10 inches

Which of the following people can apply for the job:
        a) a person who is 18 years old and 6 feet high
        b) a person who is 23 years old and 5 feet 9 inches high
        c) a person who is 62 years old and 5 feet 10 inches high

---

When there are two conditions being tested, 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 1.20**
>
> What are the other possible combinations for the two conditions?

All possibilities of the AND operator are summarised in FIG-1.14.

FIG-1.14

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 1.21**
>
> 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:

```
condition 1    OR    condition 2
```

Hence the results are determined by the following rules:

```
Determine the truth of condition 1
Determine the truth of condition 2
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:

Condition 1 (*has landed on Go to Jail*)	is	false
Condition 2 (*has thrown three consecutive doubles*)	is	true

Because at least one of the conditions is true, the overall result is true, so the player player moves to Jail.

Using the OR operator, the possible combinations and results are summarised in FIG-1.15.

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

As with AND, you can have as many OR statements as you need in a complex condition. As long as at least one of the conditions given is true, the overall result will be true.

> **Activity 1.22**
>
> In the game of Monopoly, a player can get out of jail if they throw a double (same value on both dice), pay a fine, or hand over a "Get Out of Jail Free" card.
>
> Write an IF statement that reflects this logic.

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. In standard English, the opposite of *it's dark* is *it's not dark*; in the structured English we always place the word NOT first. This means that, rather than write *it's not dark,* we would write

        NOT it's dark

In *Monopoly* a player can charge rent on a property as long as that property is not mortgaged. This situation can be described with the statement:

        IF NOT property mortgaged THEN
            Rent can be charged
        ENDIF

The NOT operator works on a single operand:

        NOT condition

When NOT is used, the result given by the original condition is reversed. Hence the results are determined by the following rules:

        1. Determine the truth of the original condition without the term NOT
        2. Complement the result obtained in step 1

For example, if a player lands on a property that is not mortgaged, then the result of the IF statement given above would be calculated as:

        Original condition (property mortgaged)   is   false
        So, since the complement of *false* is *true*, the result   is   true

This may seem a rather strange way to work out the overall result, but it will prove to be a useful approach when we examine exactly how AGK BASIC operates in a later chapter.

For many conditions you can eliminate the need for NOT by simply changing the condition. So, rather than write NOT it's dark we could write it's light (assuming light or dark are the only two options). But there are situations where using NOT will save a lot of writing. For example, it's easier to write

> IF NOT it's Monday THEN

than

> IF it's Sunday OR it's Tuesday OR it's Wednesday OR it's Thursday OR it's
> ↳Friday OR it's Saturday THEN

Although both IF statements are equivalent to each other, the first involves a lot less typing!

The results of the NOT operator are summarised in FIG-1.16.

condition	NOT condition
false	true
true	false

Mixing Boolean Operators

Conditions can get as complicated as your head can cope with. You can mix ANDs, ORs and NOTs to your heart's content in order to express the complex condition your logic requires. When you have such a complex condition, the overall result is calculated as follows:

1. Determine the truth of each condition
2. Perform all NOT operations
3. Perform all AND operations
4. Perform all OR operations

There will be some conditions where this order will not give the result you are after. For example, let's assume that to win a game you must first accumulate $100,000 and then either own 25 properties or eliminate all other players. We could write this as:

> IF player has $100,000 AND player has 25 properties OR all other players
> ↳eliminated THEN

Now, let's assume the following conditions: a player has $80,000, 18 properties and has eliminated all other players. Should this player win the game? No, because he must have at least $100,000. But if we calculate the result according to the rules above, then:

determining the truth of each condition we get:

> IF false AND false OR true THEN

Since, there are no NOT operators, we perform the AND operation giving:

> IF false OR true THEN

And, finally, performing the OR operation leaves us with:

IF true THEN

So, according to this, the player has won.

In situations like this, where we need to have the operations performed in a different order, we may use parentheses. Any operations within parentheses are always performed first. So, if we rewrite our IF statement as:

IF player has $100,000 AND (player has 25 properties OR all other players
↳eliminated) THEN

the steps become:

  IF false AND (false OR true) THEN
⇒ IF false AND true THEN
⇒ IF false THEN

and the player is shown to have not won.

Boolean operator priority is summarised in FIG-1.17.

FIG-1.17

Operator Priority

Priority	Operator
1	( )
2	NOT
3	AND
4	OR

Activity 1.23

A simple card game involves a player turning over cards from the top of a face-down deck until an Ace or a King is uncovered. At this point the game stops.

Write a WHILE statement that describes the logic involved.

Activity 1.24

a) What is another term for *condition*?
b) Give an example of a Boolean operator.
c) If two conditions are linked using the term AND, how many of the conditions must be true before the conditional statement is executed?
d) If a complex condition contains both a NOT and an AND operator, which is performed first?
e) How can we modify a Boolean expression so that an OR operator is performed before an AND operator?

Data

We know we need to retain information. Look at your phone; it is probably 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 die, where your piece is on the board and so on.

All 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 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 contain the values we have written in. These 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 AGK BASIC are:

integer   holds whole numbers only (eg -12, 0, 92).

float   holds numbers containing fractions (-14.6, 0.005, 176.0). Notice that the fraction part may be .0.

string   holds zero, one or more characters.

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

Operations on Data

There are four basic operations that an algorithm or computer program 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 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**. 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 (-). A summary of the operators available are given in FIG-1.18.

FIG-1.18

The Arithmetic
Operators

English	Symbol
Multiply	*
Divide	/
Add	+
Subtract	-

Note the standard computing practice of using * as the multiplication operator and / for division

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**. Rather than use terms such as *is less than*, we use the *less than* symbol (<). A summary of these comparison operators is given in FIG-1.19.

FIG-1.19

The Comparison
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	<>

Note the symbols <> are used to mean *not equal to*. Some languages use !=

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**.

> **Activity 1.25**
>
> 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.

Counts and Totals

Perhaps two of the commonest requirements in programming are keeping counts and keeping totals. Of course, these are operations that we perform often ourselves. If you want to deal 13 cards from a deck, you'll keep a mental count of how many you've dealt so far. If you have several items to pay for in a shop, you'll work out the total price (or maybe you'll just believe what the till tells you!).

To perform count and total operations in a detailed algorithm we need to make use of variables.

Counting

In our guessing game, we might want to know how many guesses it takes to come up with the correct value. We can do this by modifying our previous algorithm as follows:

```
Player 1 thinks of a number between 1 and 100
Set count to zero
REPEAT
    Player 2 makes an attempt at guessing the number
    Add 1 to count
    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
Player 1 states the value of count
```

The three new statements are typical of any algorithm that performs a count:

■ Initialise the count to zero.

■ Add 1 to the count each time the operation being counted occurs.

■ State the value of the count when counting has ended.

Totalling

In Activity 1.23 we described the logic of a simple card game in which cards were turned over until an Ace or a King was encountered. Let's extend this by counting the total value of the cards turned but excluding the Ace or King which terminates the process. We can perform this using the following changes to the earlier logic:

```
Set total to zero
Turn over card
WHILE NOT card is Ace AND NOT card is KING DO
    Add card value to total
    Turn over card
ENDWHILE
State value of total
```

As we can see, there is very little difference between the logic of totalling and that of counting. In fact, the only difference is the value being added to the variable.

> **Activity 1.26**
>
> Modify the card-totalling algorithm given above so that not only the total but also the number of cards turned (excluding the final card) is calculated and stated.

> **Activity 1.27**
>
> Load and run the app *CardAlgorithm.exe* which is in *AGK2/Resources/Ch01/ CardAlgorithm* of the support files you downloaded earlier.
> Note how the statements are executed and the variables change value.

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 change a flat tyre on a car as:

> Prepare car
> Remove wheel which has flat tyre
> Fit new wheel
> Ensure new tyre is at correct pressure
> Store any equipment used

However, this lacks enough detail for anyone unfamiliar with the operation. To help, we could replace the first statement Prepare car with:

> Park safely on level ground
> Switch off the engine
> Put the car in appropriate gear
> Pull on the handbrake

More detail could be added to the other original statements in the same way.

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 solution to the original problem as an equivalent sequence of tasks before going on to create a set of more detailed instructions on how to handle each of these tasks. This divide-and-conquer strategy is known as **stepwise refinement**.

Now that we've covered the idea behind stepwise refinement, let's have a look at the complete solution to creating an algorithm for changing a flat tyre:

Outline Solution:

> 1. Prepare car
> 2. Remove wheel which has flat tyre
> 3. Fit new wheel
> 4. Ensure new tyre is at the correct pressure
> 5. Store any equipment used

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

As a guideline, we should aim for a LEVEL 1 solution with between 4 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, *Prepare car* would be expanded thus:

> 1. Prepare car
>   1.1 Park safely on level ground
>   1.2 Switch off the engine
>   1.3 Put the car in appropriate gear
>   1.4 Pull on handbrake

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 step 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 of that control structure must be included in the detailed instructions for that task. Having satisfied ourselves that the more detailed breakdown is correct, we proceed to the next statement from the original solution.

```
2. Remove wheel which has a flat tyre
     2.1 Remove any hub cap
     2.2 Loosen wheel nuts
     2.3 Jack up the car
     2.4 Remove wheel
```

To fit the new wheel, the extra detail is:

```
3. Fit new wheel
     3.1 Place new wheel on car
     3.2 Replace wheel nuts
     3.3 Lower car
     3.4 Tighten wheel nuts
```

Control structures can be added where necessary. In the next breakdown we use WHILE and IF in the more detailed description:

```
4. Ensure new tyre is at correct pressure
     4.1 Check tyre pressure
     4.2 WHILE pressure not correct DO
     4.3      IF pressure too low THEN
     4.4          Pump some air into the tyre
     4.5      ELSE
     4.6          IF pressure too high THEN
     4.7              Release some air from the tire
     4.8          ENDIF
     4.9      ENDIF
     4.10     Check tyre pressure
     4.11 ENDWHILE
```

But not every statement from a level 1 solution needs to be expanded. For example, we may decide that Store any equipment used is sufficient detail for step 5, therefore no further breakdown is required.

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 Park safely on level ground
1.2 Switch off the engine
1.3 Put the car in appropriate gear
1.4 Pull on handbrake
2.1 Remove any hub cap
2.2 Loosen wheel nuts
2.3 Jack up the car
2.4 Remove wheel
3.1 Place new wheel on car
3.2 Replace wheel nuts
3.3 Lower car
3.4 Tighten wheel nuts
4.1 Check tyre pressure
```

4.2 WHILE pressure not correct DO
4.3   IF pressure too low THEN
4.4   Pump some air into the tyre
4.5   ELSE
4.6   IF pressure too high THEN
4.7   Release some air from the tire
4.8   ENDIF
4.9   ENDIF
4.10   Check tyre pressure
4.11 ENDWHILE
5.   Store any equipment used

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, Store any equipment used).

For some more complex problems it may be necessary to repeat this process to more levels before sufficient detail is achieved. For example, we might break down the statement   1.1 Park safely on level ground   to

1.1.1 Stop car out of the way of traffic on level ground
1.1.2 IF you are near passing traffic THEN
1.1.3   Turn on your hazard lights
1.1.4 ENDIF

Here a level 2 statement has been broken down into level 3 statements. To create a complete level 3 algorithm, this process would be continued for any other level 2 statements that needed to be expanded to give more detail. When complete, the appropriate statements are collected together, just as we did to create the level 2 description, to produce a level 3 breakdown.

Activity 1.28

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).

Vessels are positioned
in the left-hand grid

Results of guesses are
placed in the right-hand grid

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: O 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 O
> 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 obvious errors or omissions in the set of instructions we have created. This type of error is known as a **logic error**.

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

created earlier. 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. This set of values should be chosen so that each possible result is achieved at least once. For our game, we have three results possible each time a guess is made. These are "*Too low*", "*Too high*" or "*Correct*".

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 for our game are shown in FIG-1.20.

FIG-1.20

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. This is known as a **dry run** or **desk checking**.
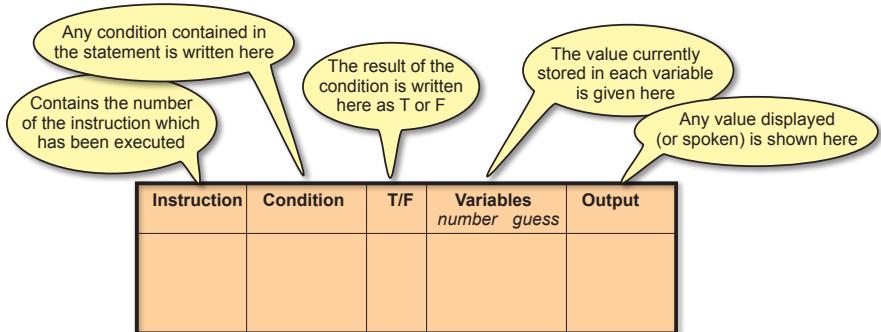
The algorithm for the number game is shown below, this time with instruction numbers added.

```
 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 table (called a **trace table**) with the headings as shown in FIG-1.21.

FIG-1.21

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-1.22.

FIG-1.22

Working through a Trace 1

Instruction	Condition	T/F	Variables number guess	Output
1			42	

The REPEAT word comes next. Although this does not cause any changes, nevertheless a 2 should be entered in the next line of our trace table. Instruction 3 involves player 2 making a guess at the number (this guess will be 75 according to our test data). After 3 instructions our trace table is as shown in FIG-1.23.

FIG-1.23

Working through a Trace 2

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

Instruction 4 is an IF statement containing a condition. This condition and its result are written into columns 2 and 3 as shown in FIG-1.24.

FIG-1.24

Working through a Trace 3

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

Because the condition is false, we now jump to instruction 6 (the ELSE line) and on to 7. This is another IF statement and our table now becomes that shown in FIG-1.25.

FIG-1.25

Working through a Trace 4

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

Since this second IF statement is also false, we move on to statements 9 and 10. Instruction 10 causes output (speech) and hence we enter this in the final column as shown in FIG-1.26.

FIG-1.26

Working through a
Trace 5

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

Now we move on to statements 11,12 and 13 as shown in FIG-1.27.

FIG-1.27

Working through a
Trace 6

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

Since statement 13 contains a condition which is false, we return to statement 2 and then onto 3 where we enter 15 as our second guess (see FIG-1.28).

FIG-1.28

Working through a
Trace 7

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

Activity 1.29

Create your own trace table for the number-guessing game and, using the same test data as given in FIG-1.20 complete the testing of the algorithm.

Were the expected results obtained?

Activity 1.30

Load and run *Remember01.exe* (part of the downloaded *Ch01* support files).

Use it to remind you of the points covered in this chapter.

Summary

- Computers can perform many tasks by executing different programs.

- An algorithm is a sequence of instructions which solves a specific problem.

- A program is a sequence of computer instructions which usually manipulates data and produces results.

- Three control structures are used in programs :

  ```
  Sequence
  Selection
  Iteration
  ```

- A sequence is a list of instructions which are performed one after the other.

- Selection involves choosing between two or more alternative actions.

- Selection is performed using the IF statement.

- There are three forms of IF statement:

  ```
  IF condition THEN
      instructions
  ENDIF

  IF condition THEN
      instructions
  ELSE
      instructions
  ENDIF

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

- Iteration is the repeated execution of one or more statements.

- Iteration is performed using one of three instructions:

  ```
  FOR number of iterations required DO
      instructions
  ENDFOR

  REPEAT
      instructions
  UNTIL condition

  WHILE condition DO
      instructions
  ENDWHILE
  ```

- A condition is an expression which is either *true* or *false*.

- Simple conditions can be linked using AND or OR to produce a complex condition.

- The meaning of a condition can be reversed by adding the word NOT.

- Data items (or variables) hold the information used by the algorithm.

- Data item values may be:

  > Input
  > Calculated
  > Compared
  > or Output

- Calculations can be performed using the following arithmetic operators:

Multiplication	*
Addition	+
Division	/
Subtraction	-

- Comparisons can be performed using the relational operators:
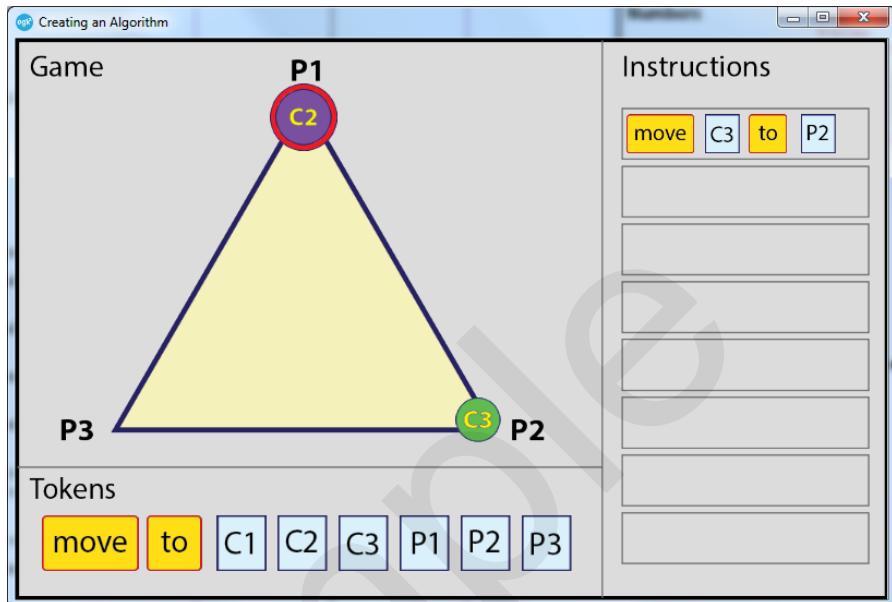
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	=
Not equal to	<>

- In programming, a data item is referred to as a variable.

- Counting involves initialising a variable to zero and incrementing it each time the event being counted occurs.

- Totalling involves initialising a variable to zero and adding a specified value to the total each time a new value is received.

- 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.

Support Material for this Chapter

Algorithm Constructor (*TriLogic.exe*)

Screen Shot

Overview

This AGK BASIC program allows you to construct and algorithm for moving the three counters (C1, C2 and C3) from position P1 to position P3. The algorithm must obey the rule that a larger counter cannot be placed on a smaller one.

User Instructions

Press the Start button (visible in the top right of the Game area when the program first starts). This moves all three counters to P1 on the playing area .

Click on the appropriate tokens to construct an instruction. All instructions should be of the form

     MOVE counter TO position

and are assembled in the Instructions area in response to the tokens selected.

If the instruction is invalid an error message will appear. Click on the error message to remove it and redo your instruction.

When a correct command is entered, the selected counter will be moved in the Game area.
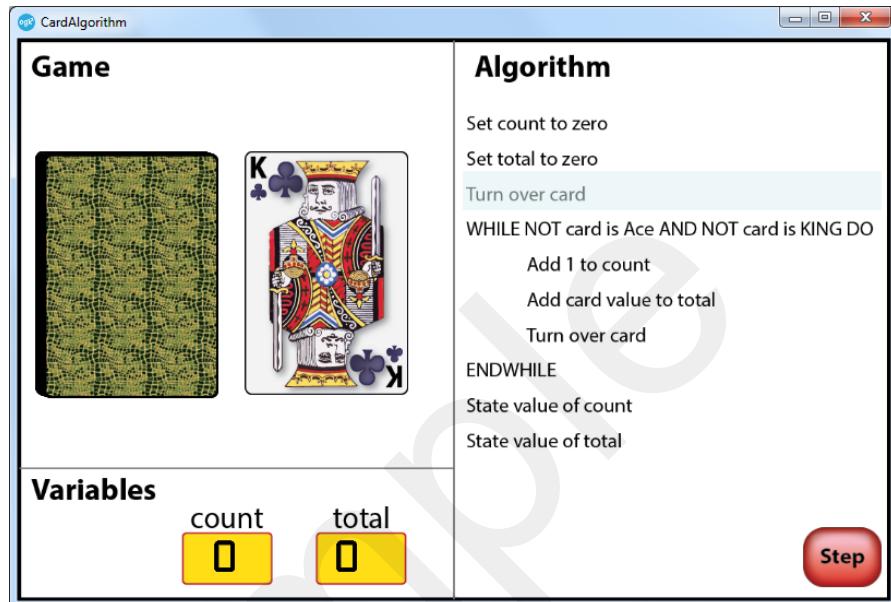
Once a move is complete, continue building instructions until all the counters are at position P3.

Download

The app file is called *TriLogic.exe* and can be found in the *AGK2/Resources/Ch01/ TriLogic* folder of the download material for this book.

Algorithm Tracer (*CardAlgorithm.exe*)

Screen Shot

Overview

This AGK BASIC program shows the step-by-step execution of an algorithm, highlighting the line which has just been executed.

The algorithm (shown in the Algorithm area of the screen) counts the number of playing cards turned over before an Ace or a King is encountered. The card turning is shown in the Game area. The total face value of those cards is also calculated. These two values are displayed in the Variables part of the screen.

The cards are shuffled each time the program is run, so the results will be different every time.

User Instructions

When the program begins, no statement is highlighted.

Press the Step button. This will execute the highlight and execute the first command.

The effect created by executing a command will result in a change in the Variables area, a card being turned in the Game area, a condition being tested, or simply the moving of the highlighter to the next line.
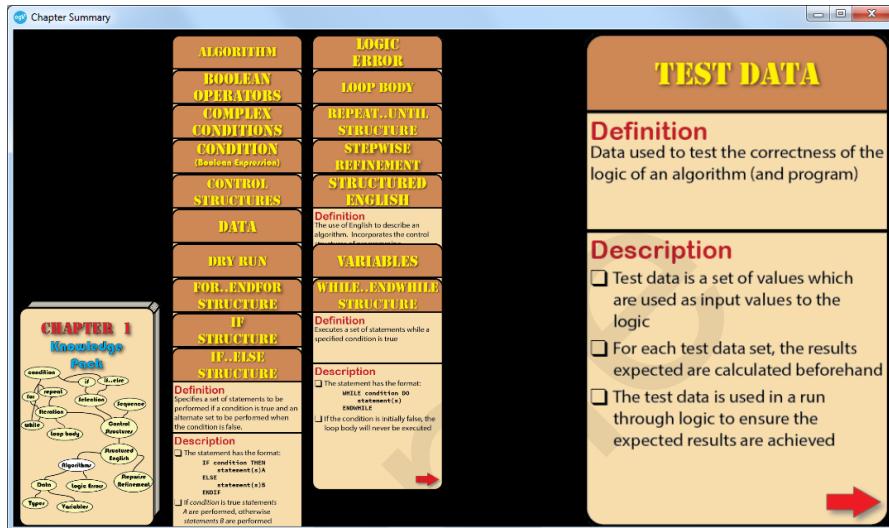
Continue to press the Step button until all the statements in the algorithm have been carried out.

Download

The app file is called *CardAlgorithm.exe* and can be found in the *AGK2/Resources/Ch01/CardAlgorithm* folder of the download material for this book.

Reviewer (*Remember01.exe*)

Screen Shot

Overview

This program displays a set of cards summarising various topics covered in this chapter.

User Instructions

When the program starts up a single card is displayed showing the main topics covered in the form of a mindmap. This is the front card of the pack.

Click on the front card. This will move the card pack to the bottom left of the screen and deal all the cards into the centre of the screen.

Click on a card title to have it expand on the right hand side of the screen.

Click on a card's arrow to see the other side of a card.

Click on a card's title to have it return to its position in the dealt pack.

Click on the front card (positioned at the bottom left) to return all the cards to the pack.

Download

The app file is called *Remember01.exe* and can be found in the *AGK2/Resources/Ch01/Remember01* folder of the download material for this book.

Solutions

Activity 1.1

No solution required.

Activity 1.2

A second solution is:

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

Activity 1.3

A possible solution to the river crossing problem:

```
Row to other side with goat
Return to first side
Row over with wolf
Return to first side with goat
Row over with cabbage
Return to first side
Row over with goat
```

Activity 1.4

A possible solution to the TriLogic game is:

```
Move C3 to P3
Move C2 to P2
Move C3 to P2
Move C1 to P3
Move C3 to P1
Move C2 to P3
Move C3 to P3
```

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

Updated algorithm for Snakes and Ladders:

```
Roll die
Move counter forward by the number thrown
IF counter stops at the bottom of a ladder THEN
   Move counter to top of ladder
ENDIF
IF counter stops at the head of a snake THEN
   Move counter to tail of snake
ENDIF
Pass die to next player
```

Activity 1.7

Algorithm for Player 2 response in Hangman:

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

Activity 1.8

```
IF guess matches number THEN          [false]
         Player 1 says "Correct"      [false]
ENDIF
IF guess is less than number THEN
         Player 1 says "Too low"      [true]
ENDIF
IF guess is greater than number THEN
         Player 1 says "Too high"
ENDIF                                 [executed]
```

Activity 1.9

```
IF guess matches number THEN          [false]
         Player 1 says "Correct"
ELSE
         IF guess is less than number THEN   [true]
             Player 1 says "Too low"
         ELSE
             Player 1 says "Too high"        [executed]
         ENDIF
ENDIF
```

Activity 1.10

```
IF W key pressed THEN
  Move character forward
ELSE
   IF S key pressed THEN
     Move character backward
   ELSE
      IF M key pressed THEN
        Move character to the right
      ELSE
         IF N key pressed THEN
           Move character to the left
         ELSE
            IF U key pressed THEN
              Make character jump up
            ELSE
               IF space key pressed THEN
                 Fire weapon
               ENDIF
            ENDIF
         ENDIF
      ENDIF
   ENDIF
ENDIF
```

Activity 1.11

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

Activity 1.12

The two statements which make up the loop body are:

```
Pick out ball
Call out number on the ball
```

Activity 1.13

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

```
FOR 10 times DO
```

Activity 1.14

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 1.15

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 1.16

```
FOR 52 times DO
   Look at card at top of deck
   IF its value is between 1 and 7 THEN
      Place in left-hand pile
   ELSE
      Place in right-hand pile
   ENDIF
ENDFOR
```

We could have started with the line

```
For each card DO
```

Activity 1.17

```
   REPEAT
      Turn over top card
   UNTIL card is an Ace
```

Activity 1.18

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

Activity 1.19

a) Cannot apply. Too young; first condition false.
b) Cannot apply. Too short; second condition false.
c) Can apply. Both conditions true.

Activity 1.20

Other possibilities are:

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

Activity 1.21

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

Activity 1.22

```
IF double thrown OR fine paid OR used "Get Out of
⤷Jail Free" card THEN
   Player gets out of jail
ENDIF
```

Activity 1.23

```
   Turn over card
   WHILE NOT card is Ace AND NOT card is KING DO
      Turn over card
   ENDWHILE
```

Activity 1.24

a) Boolean expression.
b) Boolean operators are: AND, OR, and NOT.
c) Both conditions must be true.
d) NOT has a higher priority and will be performed first.
e) By enclosing the OR operator and its operands in parenthesis, that operation will be performed before the AND.

Activity 1.25

a) Its name and value.
b) From outside the system. In a computerised setup, this is often entered from a keyboard.
c) The relational operators are:

<	(less than)
<=	(less than or equal to)
>	(greater than)
>=	(greater than or equal to)
=	(equal to)
<>	(not equal to)

Activity 1.26

Algorithm with both count and total:

```
Set count to zero
Set total to zero
Turn over card
   WHILE NOT card is Ace AND NOT card is KING DO
      Add card value to total
      Turn over card
   ENDWHILE
   State value of count
   State value of total
```

Activity 1.27

No solution required.

Activity 1.28

The LEVEL 1 is coded as:

```
1. Draw grids
2. Add ships to left grid
3. REPEAT
4.    Call grid position(s)
5.    Respond to other player's call(s)
6. UNTIL there is a winner
```

The expansion of statement 4 would become:

```
4.1 Call grid reference
4.2 Get reply
4.3 WHILE reply is HIT DO
4.4    Mark position in second grid with X
4.5    Call grid reference
4.6 Get reply
4.7 ENDWHILE
4.8 Mark position in second grid with 0
```

The expansion of statement 5 would become:

```
5.1 REPEAT
```

```
5.2     Get other player's call
5.3     IF other player's call matches position of
        ↳ship THEN
5.4        Call HIT
5.5     ELSE
5.6        Call MISS
5.7     ENDIF
5.8 UNTIL other player misses
```

Activity 1.29

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

The expected results (as shown in the Test Data table) were obtained.

Activity 1.30

No solution required.

2

Background Intel

In this Chapter:

❏ **How Number Systems Work**

❏ **The Binary System**

❏ **Converting Values between Number Bases**

❏ **Floating Point Formats**

❏ **Hexadecimal**

❏ **Negative Numbers Format**

❏ **Octal**

❏ **Storing Characters**

Number Systems

Introduction

The counting system we use today is the decimal or, more correctly, the **denary** system. It uses ten different symbols (0,1,2,3,4,5,6,7,8,9) to represent any numeric value. The number of digits used in a number system relates directly to the **base** (also known as the **radix)** of that system, hence denary is a base 10 number system.

In all modern number systems the position of a digit affects the value represented. Hence, 19 and 91, although containing the same digits, represent two different values. When we start school we are often taught the theory of our positional number system by the use of column headings:

    Thousands Hundreds Tens Units

To represent a value we merely write the required numeric symbol in each of the appropriate columns. So seven hundred and twelve is represented by placing the appropriate digit under the correct column:

    Thousands Hundreds Tens   Units
                   7   1   2

If you know a little more mathematics, then the column identities can be changed to powers of 10:

    10^3 10^2 10^1 10^0
        7   1   2

Note that the column values are based on the number system radix. So we can generalise to say that for a number system using base R, the column values can be represented by

    R^3 R^2 R^1 R^0

Computers and the Binary System

Because of the modern computer's memory design, all the information it holds, be it program instructions, numbers, text, images, sounds or video, are stored as a sequence of numeric codes.

The fundamental electronic component of computer memory is the **bit**. A bit acts rather like a light switch - it can be set to one of only two positions. Rather than consider these two positions as *off* and *on*, we treat them as numeric values, 0 and 1.

There's not much scope to store large numbers when you have only a single bit to play with, so computer memory design joins the bits into groups of 8. A group of 8 bits is known as a **byte**. Half a byte (4 bits) is sometimes known as a **nybble**.
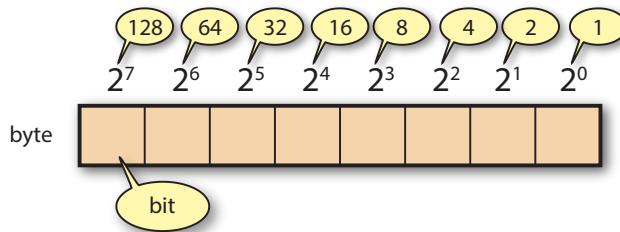
If we think of the 8 bits of a byte as the columns of a number, and knowing that each bit/column can contain only a 0 or a 1, then we can see that the computer has to make use of a base 2 number system. The term **binary** means *consisting of two parts*, and so this number system is known as the **binary number system**.

FIG-2.1 shows an abstract visual representation of a byte and the values assigned to

each column.

FIG-2.1

The Structure of a
Byte

From this we can work out that to represent the decimal value 14 in binary, we would write

128	64	32	16	8	4	2	1
				1	1	1	0

Actually, in the computer, all bits must be 0 or 1 so the actual pattern stored within a byte would include leading zeros (00001110) but for the moment we'll omit the leading zeros from our discussions on binary numbers.

Since working in more than one number base at the same time can lead to confusion, the base of a number is often added as a subscript. For example, we can state that 14 in the decimal system is written as 1110 in the binary system with the line

$$14_{10} \quad = \quad 1110_2$$

Converting From Decimal to Binary

If we want to convert a decimal number to binary, we need to work out what numbers from the binary system column values add up to our decimal number. In the case of, for example, the value 23, it can be constructed from $16 + 4 + 2 + 1$ (all binary column values). So we can say that 23 is written in binary as

128	64	32	16	8	4	2	1
			1	0	1	1	1

In other words,

$$23_{10} \quad = \quad 10111_2$$

Although this approach is easy enough to use for small numbers, it gets a bit harder with larger values, so a more methodical approach to converting decimal values to binary is required.

That approach involves dividing the decimal number to be converted by 2, writing down the quotient and remainder, then continuing the process, always dividing the quotient by 2. When the quotient reaches zero, the remainders are written in reverse order (latest one first) and this gives us the binary equivalent of the original decimal value.

FIG-2.2 shows how this method is used to convert 13_{10} to binary.
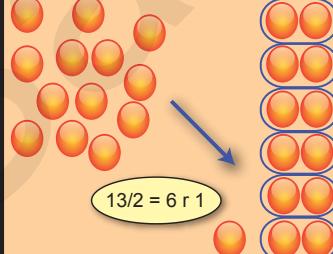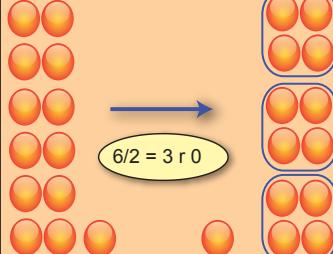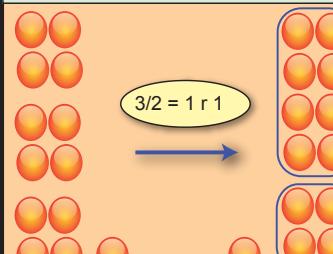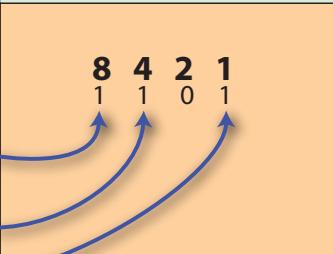
FIG-2.2

Converting Decimal
to Binary

We start by dividing the decimal number we want to convert by 2, writing down the quotient and remainder.	Next, we divide the quotient by 2, again writing down the new quotient and remainder.

$2 \lfloor 13$
    6 r 1
  quotient *remainder*
	$2 \lfloor 13$
	$2 \lfloor 6$ r 1
	    3 r 0

The process is repeated until the quotient is zero.	Finally, by rewriting the remainders obtained in reverse order, we get the binary equivalent of the original value.

$2 \lfloor 13$
$2 \lfloor 6$ r 1
$2 \lfloor 3$ r 0
$2 \lfloor 1$ r 1
    0 r 1
	$2 \lfloor 13$
	$2 \lfloor 6$ r 1
	$2 \lfloor 3$ r 0
	$2 \lfloor 1$ r 1
	    0 r 1 → **1101**
	  Binary equivalent

Perhaps it is not blindingly obvious why this approach gives us the result we are after but if we go through the process again using 13 pebbles, then it should become clearer (see FIG-2.3).

FIG-2.3

Why Division by 2
Works

If we start with 13 pebbles and group them into pairs, we are left with 6 groups of 2 and 1 remaining pebble.	If we now take two pairs and group these, we get 3 groups of four as well as our single pebble.
13/2 = 6 r 1	6/2 = 3 r 0
Pairing off our groups of 4 we get one group of 8 , a remaining group of 4 and the single pebble.	Each group size is a power of 2 (1, 4 and 8). These sizes tell us which binary columns should contain a 1.
3/2 = 1 r 1	**8 4 2 1** 1 1 0 1

Converting from Binary to Decimal

To convert from binary to decimal, take the value of each column that contains a 1 and add these values together to get the decimal equivalent (see FIG-2.4).

FIG-2.4

Binary to Decimal

$$2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$
$$0 \quad\; 0 \quad\; 0 \quad\; 0 \quad\; 1 \quad\; 1 \quad\; 0 \quad\; 1$$

$$1 \times 2^0 \;=\; 1$$
$$1 \times 2^2 \;=\; 4$$
$$1 \times 2^3 \;=\; \underline{8}$$
$$13$$

Converting Fractions

Decimal Fractions to Binary

So far we have looked at converting only whole numbers but we also need to be able to represent decimal fractions in binary.

In the decimal system, column values to the right of the decimal point are

$$10^{-1} \quad 10^{-2} \quad 10^{-3} \quad \text{etc.}$$

alternatively written as

$$\frac{1}{10} \quad \frac{1}{100} \quad \frac{1}{1000}$$

Column values to the right of the binary point, on the other hand, have the values

$$2^{-1} \quad 2^{-2} \quad 2^{-3}$$

or

$$\frac{1}{2} \quad \frac{1}{4} \quad \frac{1}{8}$$

or
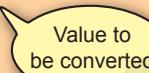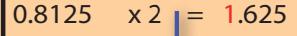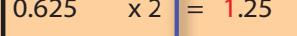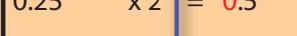
0.5   0.25   0.125

To convert a decimal integer, we had to find how that number could be expressed in terms of values which were exact powers of 2. For example, we saw that 13 could be expressed as $8 + 4 + 1$ ($2^3 + 2^2 + 2^0$). To convert a decimal fraction to a binary fraction we need to find the same thing; powers of 2 that express the original value. The only difference this time is that the power value will be negative since the values involved are less than one. For example, the value 0.625 can be expressed as $0.5 + 0.125$ ($2^{-1} + 2^{-3}$).

This time the standardised conversion process involves continually multiplying fractions by 2 and eliminating the integral part after each multiplication. The steps for converting 0.8125_{10} to binary are shown in FIG-2.5.

FIG-2.5

Converting Decimal Fractions to Binary

We start by multiplying the fraction to be converted by 2.	The fractional part of the result is then multiplied by 2.
0.8125 x 2 = 1.625 (Value to be converted)	0.8125 x 2 = 1.625 0.625 x 2 = 1.25
Again the fractional part of the result is multiplied by 2. This process continues until the fractional part is zero or the degree of accuracy required is reached.	Now, we start with the binary point and then read the integral parts of each result (starting at the top) to find the binary equivalent
0.8125 x 2 = 1.625 0.625 x 2 = 1.25 0.25 x 2 = 0.5 0.5 x 2 = 1.0 (Zero)	0.8125 x 2 = 1.625 0.625 x 2 = 1.25 0.25 x 2 = 0.5 0.5 x 2 = 1.0 .1101$_2$

When we first multiply by 2, if the number to be converted is 0.5 or greater, we will get a 1 in the integral part of the answer. This tells us that there was a 0.5 component in the original number. By ignoring that integral part in the next calculation we have, in effect, removed the 0.5 value from our calculation. The next multiplication (at this point we have multiplied by 4 — multiplication by 2 twice), causes any 0.25 component in the original value to produce an integral of 1. This integral is then ignored in the next calculation.

Unlike integer values, when we convert a decimal fraction to a binary one, we have no guarantee that the binary fraction we produce will be exactly equivalent to the decimal fraction. In these cases, we must convert to the required number of binary places.
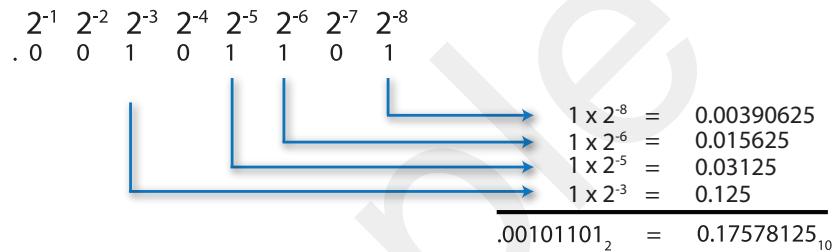
To convert a number which has integral and fractional parts, convert each part separately as if they were two separate values.

Binary Fractions to Decimal

Binary fraction to decimal does not require a new approach. It is handled in exactly the same way as binary whole numbers were converted. The only difference in this case is the column values involved (see FIG-2.6).

FIG-2.6

Converting Binary
Fractions to Decimal

$$2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}\ 2^{-5}\ 2^{-6}\ 2^{-7}\ 2^{-8}$$
$$.\ 0\quad 0\quad 1\quad 0\quad 1\quad 1\quad 0\quad 1$$

$$1 \times 2^{-8}\ =\ 0.00390625$$
$$1 \times 2^{-6}\ =\ 0.015625$$
$$1 \times 2^{-5}\ =\ 0.03125$$
$$1 \times 2^{-3}\ =\ 0.125$$

$$.00101101_2\ =\ 0.17578125_{10}$$

Hexadecimal

Hexadecimal (often shortened to **hex**) is another number system widely used in computing. This number system has the base 16 so it follows that there must be 16 different digits used in the representation of hexadecimal values.

These 16 digits are formed from the 10 digits of the decimal system and the first 6 letters of the alphabet So in hexadecimal 0 to 9 represents the normal decimal values 0 to 9 but A represents 10, B 11, C 12, D 13, E 14 and F 15. With a base 16 number system the column values for this system are:

$$16^3 \qquad 16^2 \qquad 16^1 \qquad 16^0$$
or
$$4096 \quad 256 \quad\ 16 \qquad 1$$

Converting from Decimal to Hexadecimal

Conversion from decimal to hexadecimal is little different from the conversion of decimal to binary. However, this time, instead of dividing by 2, we divide by 16. FIG-2.7 shows how the number 131_{10} is converted to hexadecimal.

FIG-2.7
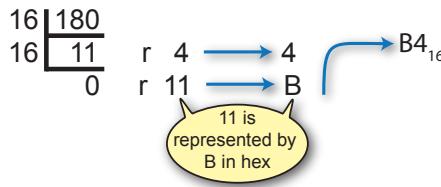
Converting Decimal
to Hexadecimal 1

```
16 | 131
16 |   8    r  3        83₁₆
        0    r  8
```

Things are slightly more complicated if the remainder is greater than 9, since we have to remember to convert the remainder to the appropriate hexadecimal letter (see FIG-2.8).

FIG-2.8

Converting Decimal to Hexadecimal 2

$$
\begin{array}{r}
16 \mid 180 \\
16 \mid \underline{11} \quad \text{r } 4 \longrightarrow 4 \\
0 \quad \text{r } 11 \longrightarrow B
\end{array}
\longrightarrow B4_{16}
$$

11 is represented by B in hex

Activity 2.6

Convert the following decimal values to hexadecimal:

a)   97   b)   212   c)   255

Converting from Hexadecimal to Decimal

Again the conversion from hexadecimal to decimal is similar to that for binary to decimal; the value in a given column is multiplied by the column value. The only change this time is that a wider range of digits may appear in any one column and that the column values are different.

FIG-2.9 shows the value $B4_{16}$ being converted to decimal.

FIG-2.9

Converting Hexadecimal to Decimal

$$
\begin{array}{cc}
16^1 & 16^0 \\
B & 4
\end{array}
$$

$$
\begin{aligned}
4 \times 16^0 &= 4 \\
B \times 16^1 &= \underline{176} \\
& \ 180
\end{aligned}
$$

B_{16} is 11_{10} so the calculation is $11_{10} \times 16_{10}$

Activity 2.7

Convert the following hexadecimal values to decimal:

a)   2C   b)   A6   c)   DE

Converting from Binary to Hexadecimal

Although the computer uses binary for everything it does, we humans find that number system a bit long-winded since it takes a large number of digits to represent even relatively small values. The other major problem we have with binary is trying to copy out values correctly; with only 0's and 1's, its all too easy to make a mistake when copying so many digits.

Using hexadecimal gives us a way of avoiding writing values in binary and yet, at the same time, making it easy to convert to and from binary when necessary.
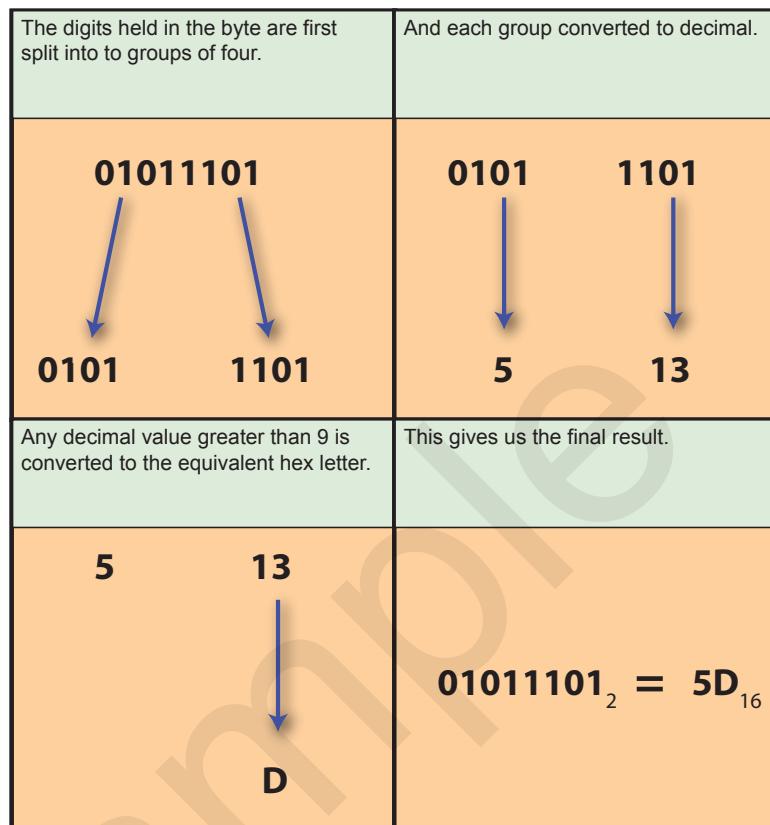
In hexadecimal, **F** is the highest value digit being equal to 15_{10}. Now 15 also happens to be the maximum value that can be represented in the four bits of a nybble. This

means that one hexadecimal digit can be used to represent four bits; two hexadecimal digits can represent a byte.

FIG-2.10 shows the steps involved in converting the 8 bits of a byte to hexadecimal.

FIG-2.10

Converting Binary to Hexadecimal

The digits held in the byte are first split into to groups of four.	And each group converted to decimal.
01011101 **0101** **1101**	**0101** **1101** **5** **13**
Any decimal value greater than 9 is converted to the equivalent hex letter.	This gives us the final result.
5 **13** **D**	$01011101_2 = 5D_{16}$

Activity 2.8

Convert the following binary values to hexadecimal:
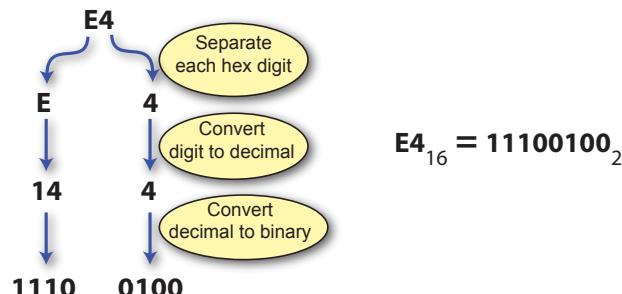
a)   01000111   b)   11111111   c)   11001011

Converting from Hexadecimal to Binary

This is nothing more than a reverse of the previous process. Each hexadecimal digit is converted to decimal and then to a 4 bit value. The process is shown in FIG-2.11.

FIG-2.11

Converting Hexadecimal to Binary

E4

E   4   Separate each hex digit

14   4   Convert digit to decimal

1110   0100   Convert decimal to binary

$E4_{16} = 11100100_2$

Octal

Octal is a base 8 number system using the digits 0 to 7. At one time some computers had their memory organised into 6 bit blocks rather than the usual 8. Just as hexadecimal is used as a convenient way of representing 4 bits, so octal was used to represent 3 bits. Two octal digits specifying the contents of 6 bits. Although little used now, octal is included here simply for completeness and because there are options in AGK BASIC to allow octal values to be used.

Since octal uses the base 8, column values would be:

$$8^4 \qquad 8^3 \qquad 8^2 \qquad 8^1 \qquad 8^0$$

To convert binary to octal we split the binary value into groups of 3 bits (starting from the right) then convert each group to its octal equivalent. This process is reversed to convert octal to binary.
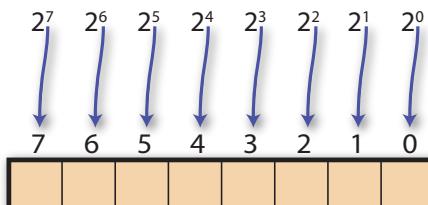
Storing Numbers

The bits within a byte are identified in diagrams by allocating the exponent value of the appropriate number column (see FIG-2.12).

FIG-2.12

Identifying the Bits within a Byte

The right-hand bit (bit 0) is known as the **least-significant bit** since it is of the least numeric value; the left-hand bit is the **most-significant bit**.
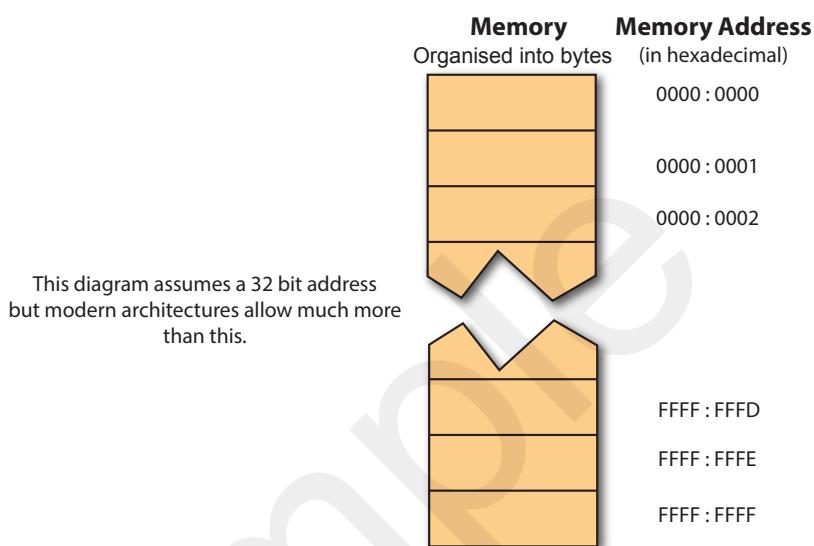
Using a single byte, values in the range 0_{10} (00000000_2) to 255_{10} (11111111_2) can be stored.

When a whole number is stored in a computer's memory, it will normally occupy one, two, four or even eight bytes. The more bytes that are allocated to it, the larger the range of values that can be stored.

Memory is designed in such a way that every byte is allocates a unique address (just like every house in a street has its own unique address). This means that the computer can directly access any byte by specifying the address of that byte (see FIG-2.13).

FIG-2.13

Memory Organisation

Memory
Organised into bytes

Memory Address
(in hexadecimal)

This diagram assumes a 32 bit address but modern architectures allow much more than this.

0000 : 0000

0000 : 0001

0000 : 0002

FFFF : FFFD

FFFF : FFFE

FFFF : FFFF

The individual bits within a byte cannot be directly accessed but other methods are available to determine the contents of any single bit.

When using multiple bytes to store a value, the right-most byte is known as the **least-significant byte**; the left-hand one, the **most-significant byte**.

Using eight bytes, numbers between 0 to 18,446,744,073,709,551,615 can be stored.

It should be noted that in reality it is common practice to store the bytes of a number in reverse order when written to computer memory (see FIG-2.14).

FIG-2.14

How an Integer is stored in Memory

The bytes of an integer value (shown here in hexadecimal)....	...are often stored in computer memory in reverse order because of the hardware design.
0X170ABEFF	Memory address FF 001C:1A00 BE 001C:1A01 0A 001C:1A02 17 001C:1A03

Despite the apparently strange storage format, the hardware automatically handles the storing and retrieval of the data, ensuring that the original value is always presented unchanged to any program accessing the value.
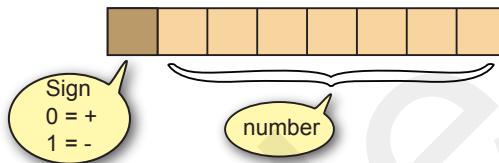
Negative Numbers in Binary

From what we've seen so far, binary can be used to store huge positive values, but how can we store a negative number such as -17?

Remembering that absolutely everything in a computer is stored in 0's and 1's we have to be imaginative with our approach to representing negative values. For example, when storing the number in a single byte, we could reserve one bit to be the sign of the number with a 0 representing + (positive) and a 1 representing - (negative) (see FIG-2.15).

FIG-2.15

Format for Storing Signed Values

At first glance, this might seem to solve our problem. We can represent +7 as 00000111 and -7 as 10000111. Of course, because there are now only 7 bits available for the magnitude of the number, we're limited to a range from -127 (11111111) to +127 (01111111). Another curious thing about this approach is that we have two ways of representing zero: 00000000 (+0) and 10000000 (-0).

But the biggest problem with this approach is that adding a negative number to a positive number doesn't produce the correct results. For example, we know that

$$9 + (-7) = 2$$

But when we attempt the same thing with our binary values

```
00001001 (+9)
10000111 (-7)
------------
10010000 (-16)
```

we arrive at an incorrect result.

One's Complement

Another possible way of representing a negative value is to make the digits exactly the opposite of the positive form. So if +7 is written as

```
00000111
```

then -7 becomes

```
11111000
```

> **Activity 2.12**
>
> Write down the equivalent of the decimal value -248 using the above system.

We have to be careful not to confuse ourselves if we use this approach.

Of course, we cannot allow a situation where a particular binary pattern — in this

case 00000111 — can have an ambiguous value (7 or -248). The solution to this is again to limit the magnitude of the values allowed. If positive value can only go as high as 127 then bit 7 (the left-most bit) will always be zero, so when we complement all the bits to represent a negative value, the left-most bit will always be a one. So, again, the left-most bit turns out to be our sign bit. When it contains a zero we have a positive number; when it contains a 1, we have a negative value.

This way of representing negative values is known as **1's complement form**.

So let's see if 9 + (-7) gives the correct result this time:

```
    00001001 (+9)
    11111000 (-7)
  ------------
1   00000001
```

Notice that the result is 9 bits long, not 8. But this extra, left-hand bit (known as the **overflow bit**) is of no consequence to us since there is no room to store it in a single byte, so it can be eliminated from our result. This leaves us with the value

```
    00000001 (+1)
```

This time the result is just 1 out from the correct answer.

> **Activity 2.13**
>
> Using 1's complement form for the binary values, calculate the result obtained when performing the following addition 14 + (-5).

In fact, using 1's complement always gives us a result that is exactly 1 less than the true result. Well, that's easily fixed — we just have to do our calculation and then add 1 to whatever result we get.

Two's Complement

However, another way of dealing with the problem is to add that extra 1 to the negative value before you start the calculation. This is known as **2s complement form**. So -7 is represented in 2's complement form by

```
    11111000  (-7 in 1s complement form)
         +1
    --------
    11111001
```

This time when we do our calculation 9 + (-7) we get

```
    00001001 (+9)

    11111001 (-7)
  ------------
    00000010 (2)
```

At last we have the correct result.

> **Activity 2.14**
>
> Using 2's complement form, redo the calculation 14 + (-5).

Notice that the left-most bit of the value still acts as a sign bit: 0 when a positive value

is stored, 1 for a negative value.

Two's complement allows values in the range -128 to + 127 to be stored in a single byte.

-128_{10} is stored as 10000000_2.

When 2s complement form is used to store a value over two bytes, it can store values in the range -32,678 to +32,767.

Many programming languages (but not AGK BASIC) allow numbers to be stored in either **unsigned** (zero and positive numbers only) format or **signed** (negative, zero and positive values) format.

In unsigned format all the bits assigned hold the number's value giving a large range of positive values, but negative numbers cannot be stored. Signed format uses 2's complement, effectively allocating 1 bit for the sign bit with the remaining bits recording the value. Although this allows negative numbers to be recorded, it halves the largest possible value that can be stored.

Floating Point Values in Binary

The E notation is most often seen on a calculator or computer. When written by hand, the E is usually replaced by x10.

In the decimal system, real numbers (those with a decimal point) can be expressed in a different way from whole numbers. On paper these type of values are either written using **fixed point notation** with a fixed number of digits after the decimal point (e.g. 128.3 — one digit after the decimal point) or in **scientific notation** (e.g. 1.283E2).

Scientific notation may look a little strange if you haven't come across it before, but it is really quite easily understood:

The letter **E** stands for the term **exponent** or **x10 raised to the power**
The number to the left of the E is called the **significand**.
The number to the right of the E is the **exponent value**.

The exponent represents the power to which 10 is to be raised. So E01 means 10^1 or simply 10; E02 means 10^2 or 100.

To arrive at the number being represented, we perform the calculation

significand x $10^{exponent}$

so

1.283E2

= 1.283 x 10^2

= 1.283 x 100

= 128.3

Where we have a negative exponent such as 10^{-1} or 10^{-2} then these represent 1/10 (0.1) and 1/100 (0.01) respectively. So a number shown in scientific notation as

1.67E-3

is

$$1.67 \times 10^{-3}$$
$$\Rightarrow \quad 1.67 \times 0.001$$
$$\Rightarrow \quad 0.00167$$

Of course, we might have written the value 128.3 as .1283E03 or 12.83E1 or 1283E-01 but the convention is to make sure that the integral part of the significand lies in the range 1 to 9. When the significand is within this range, it is termed a **normalised significand**.

For values less than 1, normalising the significand will mean that the exponent will be negative. For example,

$$0.000013$$
$$\Rightarrow \quad 1.3E-5$$

If the number being represented is negative, then the significand is negative:

$$-382.19$$
$$\Rightarrow \quad -3.8219E2$$

Activity 2.15
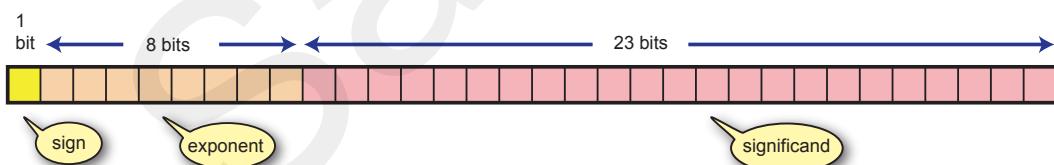
Rewrite the following values in standard notation:

a)   8.7512E3   b)   -3.8122E2   c)   6.1937E-2

Most software uses the significand-exponent approach for storing real numbers in a format known as **floating-point** but, of course, the significand and exponent are held as binary values.

The standard layout for a 32 bit floating point value is shown in FIG-2.16.

FIG-2.16 Storing Floating Point Values

The exact formatting on the computer is slightly different from the decimal version.

*The exponent is said to have a **bias** of 127.*

- The exponent is always 127 greater than its true value. This is simply a method of eliminating the need to use 2's complement when the true exponent is negative.

- The significand is adjusted so its value lies between 1 and 2.

- The sign bit is 0 if the number is positive; 1 if the number is negative.

The significand itself is always stored in positive form even when the value represented is negative; 2's complement is not used. Also, the leading 1 of the significand is not actually stored in memory; instead, its presence is assumed when the computer performs any subsequent calculations.

Let's see how the number 28.75 would be stored in this format:

$$28.75_{10} = 11100.11_2$$

Normalising the binary mantissa we get:

1.110011 E 100

The leading 1 of the significand is assumed, so only

110011   is stored.

Adding 127 to the exponent gives

10000011

So, when stored in the 32 bit format shown above we get

0   10000011   11001100000000000000000

Activity 2.16

a)   Convert the value 0.00872 to binary floating point format.

b)   A floating point formatted value is held within the computer as

1   10000001   01011010000000000000000

Calculate the decimal equivalent of this value.

There are two main situations where a floating point value is interpreted differently by the software:

- When the exponent is zero and the significand is zero, the value held is assumed to be zero.

- When the exponent has the value 11111111_2 (FF_{16}) and the significand is zero, this represents infinite. When the exponent is FF_{16} but the significand is not zero, this represents an error condition and is often shown in program output as NaN (**N**ot **a** **N**umber).

Note that when using 64 bits to store a floating point value, the leading 1 in the significand is actually stored rather than assumed.

Character Coding

As well as numbers, computers need to store characters. Since everything within the machine is stored in binary, this means that we need some sort of coding system to represent characters.

The most universally used coding system in the past has been American Standard Code for Information Interchange (**ASCII** — pronounced *ask - ay*). This uses a single byte to store a character with codes for upper and lower case letters, punctuation marks, numeric digits and a few other symbols.

Only 7 of the 8 bits in a byte are used for the character, the 8th bit originally being

used as a parity bit to help with detection of errors produced during the transmission of data.

In ASCII a capital A is coded as 01000001; a B as 01000010, etc.

The ASCII coding system is quite restrictive with no scope for representing non-European characters.

To cope with the wider range of characters, the **Unicode Standard** was created which has assigned a unique code to every possible character (over 100,000). This coding convention is now used by all modern software.

The most widely used of the Unicode character-coding systems is **UTF-8** (**U**nicode **T**ransformation **F**ormat -8 bit). This uses a variable number of bytes for coding characters. For the original ASCII character set, UTF-8 uses a single byte, employing exactly the same codes as ASCII. When other characters such as those from Greek, Hebrew and Arabic are used, UTF-8 uses two bytes per character. Most other language characters (Chinese, Japanese, etc) require three bytes of coding. More specialised symbols (some mathematical and historic scripts) make use of a fourth byte.

Characters can also be coded using UTF-16 (which codes in characters in either 2 or 4 bytes as required) and UTF-32 (which codes all characters using 4 bytes).

Summary

- A modern number system's base or radix is determined by the number of different symbols used to represent characters.

- A number system's column values are the radix value raised to incrementing powers (increments right to left).

- Binary is a base 2 number system using the digits 0 and 1 to represent all values.

- An individual binary digit is known as a bit.

- A grouping of 4 bits is known as a nybble.

- A grouping of 8 bits (2 nybbles) is known as a byte.

- The base of a number is often included as a subscript where confusion might otherwise arise.

- Integer decimal values are converted to binary by continually dividing by 2 until the quotient is zero and then writing out the remainders in reverse order.

- An integer binary value can be converted to decimal by summing the value of all columns containing a 1.

- Decimal fractions are converted to binary by continually multiplying the fractional part of each result by 2 then listing the integral parts of the results.

- Binary fractions are converted to decimal by summing the values of all columns containing a 1.

- Hexadecimal is a base 16 number system using the digits 0-9,A-F.

- A single hexadecimal digit is a convenient way of representing 4 bits.

- Decimal can be converted to hexadecimal by continual dividing by 16 and

writing out the remainder (last one first). Any remainders of 10 or more must be converted to a hexadecimal letter.

■ A hexadecimal value can be converted to decimal by multiplying each hex digit by the value of the column in which it is positioned and then summing the results.

■ To convert binary to hexadecimal, split the binary value into groups of four bits, convert the four bits to decimal then convert any values greater than 9 to a hexadecimal letter.

■ To convert from hexadecimal to binary, convert each hexadecimal digit to exactly 4 bits.

■ Octal is a base 8 number system using the digits 0 - 7.

■ A single octal digit is used to represent 3 binary digits.

■ Negative integer values are stored in 2s complement form.

■ A negative number's 2's complement form is derived by taking the binary form of a positive number, inverting all the digits and adding 1.

■ The computer uses floating-point format to store real numbers.

■ Floating point format has three components:
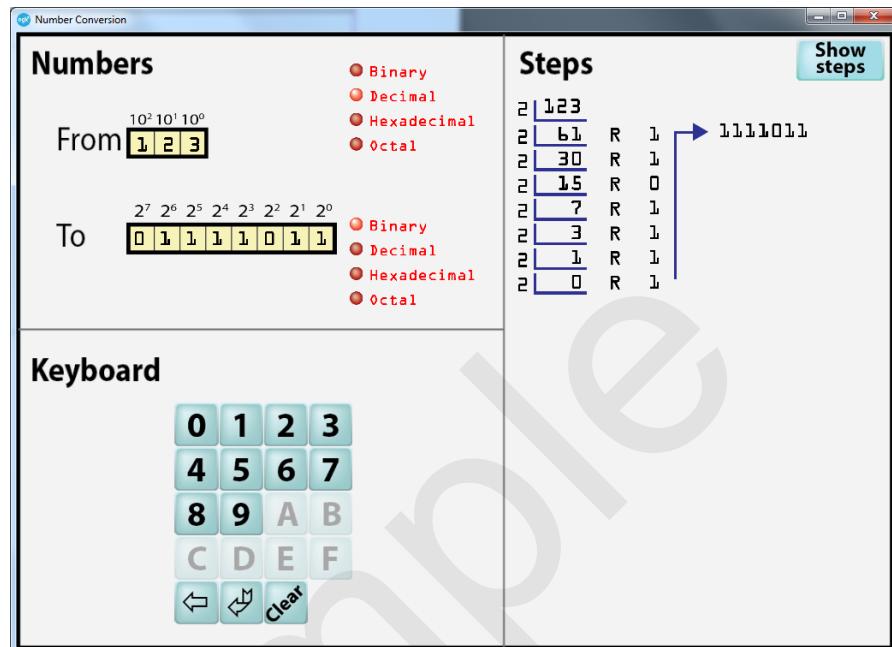
   sign bit
   exponent
   significand

■ The sign bit is 0 for positive values; 1 for negative values.

■ The significand of a floating point value is always stored in its positive form even when the value represented is negative.

■ The exponent has a bias (or offset) of 127 removing the need to store negative values.

■ When using 32 bits to store a real value, the significand is normalised to assume a leading 1 which is not stored.

■ When using 64 bits to store a real value, the leading 1 in the normalised significand is stored.

■ Characters can be coded in a single byte using ASCII format.

■ The most widely used character coding system now in use is UTF-8 which uses a variable number of bytes per character.

■ UTF-8 uses the same single-byte coding as ASCII for the ASCII character set.

Support Material for this Chapter

Integer Number Converter (*Numbers.exe*)

Screen Shot

Number Conversion

Numbers

○ Binary
○ Decimal
○ Hexadecimal
○ Octal

From $10^2\ 10^1\ 10^0$ | 1 | 2 | 3 |

To $2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$ | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

○ Binary
○ Decimal
○ Hexadecimal
○ Octal

Steps   [Show steps]

```
2 | 123
2 |  61   R   1        1111011
2 |  30   R   1
2 |  15   R   0
2 |   7   R   1
2 |   3   R   1
2 |   1   R   1
2 |   0   R   1
```

Keyboard

```
0  1  2  3
4  5  6  7
8  9  A  B
C  D  E  F
⇦  ⤷  Clear
```

Overview

This AGK BASIC program allows you to convert a positive integer number from one base to another and shows how the conversion would be achieved manually.

User Instructions

In the Numbers Panel:

Select the *From* number base using the top set of radiobuttons. The keyboard will highlight only the keys appropriate to that number base

Select the *To* number base using the bottom set of radiobuttons.

In the Keyboard Panel:

Type in the value you want to convert. You can use the ⇦ button to delete the last digit or the **Clear** button to delete all digits entered.

Press the ⤷ button to enter your completed value. The value you entered will automatically be converted to the other number base and displayed in the *To* area.

In the Steps Panel:

Press the **Show Steps** button to reveal how the conversion between the selected number bases would be performed manually.

Error Messages

You'll get an error message if you try to enter a value too large to be stored in a single byte.

You'll get an error message if you try to display the steps required to convert directly from Octal to Hexadecimal (or vice versa) since there is no direct manual conversion method. Conversion between these two bases is usually performed by converting to binary as an intermediate step.

Download

The app file is called *Numbers.exe* and can be found in the *AGK2/Resources/Ch02/Numbers* folder of the download material for this book.

Solutions

Activity 2.1
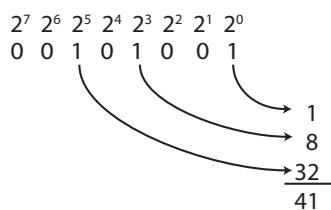
a) Working:

```
2 | 19
  2 | 9    r    1  ─┐→ 10011
    2 | 4    r    1  │
      2 | 2    r    0  │
        2 | 1    r    0  │
          | 0    r    1 ─┘
```

a) 19_{10} $= 10011_2$

b) 72_{10} $= 1001000_2$

c) 63_{10} $= 111111_2$

Activity 2.2

a) Working:

$$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$$
$$0\ \ 0\ \ 1\ \ 0\ \ 1\ \ 0\ \ 0\ \ 1$$

```
                          → 1
                        → 8
                      →32
                       41
```

a) 00101001_2 $=$ 41_{10}

b) 11111111_2 $=$ 255_{10}

c) 10101010_2 $=$ 170_{10}

Activity 2.3

The largest value that can be stored in a nibble is $1111_2 = 15_{10}$.

Activity 2.4

0.3125	x 2	=	0.625
0.625	x 2	=	1.25
0.25	x 2	=	0.5
0.5	x 2	=	1.0

0.3125_{10} $=$ $.0101_2$

Activity 2.5

$0.100101_2 = 0.5 + 0.0625 + 0.015625$ $= 0.578125_{10}$

Activity 2.6

a) Working:

```
16 | 97
  16 | 6       r    1
     | 0       r    6
```

a) 97_{10} $=$ 61_{16}

b) 212_{10} $=$ $D4_{16}$

c) 255_{10} $=$ FF_{16}

Activity 2.7

a) $2C_{16}$ $=$ 44_{10}

b) $A6_{16}$ $=$ 166_{10}

c) DE_{16} $=$ 222_{10}

Activity 2.8

a) 01000111_2 $=$ 47_{16}

b) 11111111_2 $=$ FF_{16}

c) 11001011_2 $=$ CB_{16}

Activity 2.9

a) AB_{16} $=$ 10101011_2

b) $8C_{16}$ $=$ 10001100_2

c) $9A_{16}$ $=$ 10011010_2

Activity 2.10

a) 147_{10} $=$ 235_8

b) 75_8 $=$ 61_{10}

Activity 2.11

a) $101110_2 = 56_8$

b) 34_8 $=$ 011100_2

Activity 2.12

We start by converting the positive value 248_{10} to binary. This gives us 11111000_2. Now the bits are complemented, and we get 00000111.

Activity 2.13

```
14                =    00001110
-5                =    11111010     (1's complement form)
adding gives           00001000     (+8)
```

Activity 2.14

```
14                =    00001110
-5                =    11111011     (2's complement form)
adding gives           00001001     (+9)
```

Activity 2.15

a) 8751.2

b) -381.22

c) 0.061937

Activity 2.16

a)

$0.00872_{10} = $ `0 01111000 10001110110111100101010`

The significand is calculated as follows:

```
0.00872 x 2 = 0.01744
0.01744 x 2 = 0.03488
0.03488 x 2 = 0.06976
0.06976 x 2 = 0.13952
0.13952 x 2 = 0.27904
0.27904 x 2 = 0.55808
```

0.55808 x 2 = 1.11616
0.11616 x 2 = 0.23232
0.23232 x 2 = 0.46464
0.46464 x 2 = 0.92928
0.92928 x 2 = 1.85856
0.85856 x 2 = 1.71712
0.71712 x 2 = 1.43424
0.43424 x 2 = 0.86848

etc.

Hence the significand starts   .00000010001110

Normalising this, we get   1.0001110

This means the exponent is -7
Adding a bias of 127 we get an exponent of 120
$120_{10} = 01111000_2$

The leading 1 in the mantissa is assumed, so it is stored beginning with the digits   0001110.

b)

1 10000001 01011010000000000000000 = 5.40625

The exponent is 129
Subtracting the 127 bias, we get an exponent of 2.

Adding the assumed 1. to the significand, we get

1.0101101

Adjusting for the exponent, we get 101.01101

The integral part of this 101 is 5.

The fractional part is:

0.25 + 0.125 + 0.03125

=   0.40625

Since the sign bit is zero, the number is a positive one and, expressed in decimal, is:

5.40625

3

Starting AGK2 BASIC

In this Chapter:

❏ **Understanding Compilation**

❏ **Getting Started with AGK2**

❏ **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.

There are many programming languages; C, C++, Java, C#, and Javascript 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.

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

We are going to use a programming language known as AGK2 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, AGK2 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

For a moment, let's forget about AGK2 BASIC and consider the steps that occur when we write a program in another language such as C++ and want to run that program on our computer.

When we write a program in a language, the statements we use 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 computer only understands instructions given as 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 C++.

However, this need not be a problem; we simply need to translate the C++ 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 C++ statements which we enter into the computer using an Integrated Development Environment (IDE).

The IDE acts not only as text editor allowing program statements to be typed in and edited, but also performs all the steps required to convert the original C++ code into a form which can be executed. The translator (known as a **compiler**) is part of the IDE. After typing in our program instructions these are **compiled** to produce the equivalent instructions in machine code.

The original program code is known as the **source code**; the machine code is known as the **object code** and is saved to create an **executable file**.

The object code can be executed from within the IDE or independently by loading the executable file. The machine code instructions are then executed by the computer and we should see the results of our logic appear on the screen (assuming there are output statements in the program).
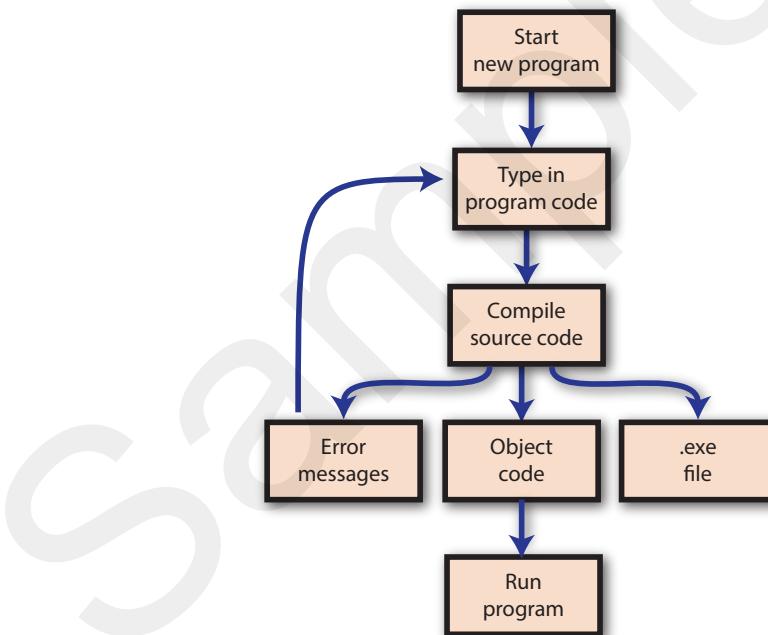
The compiler is a very exacting task master. The structure, or **syntax**, of every statement must be exactly right. If you make the slightest mistake, even something as simple as missing out a comma or misspelling a word, the translation process will fail. When this happens, a window appears giving 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 and the compilation process finished, the executable file is produced. This new file (which has an *.exe* extension), contains a copy of the object code. We can run the program by selecting the *Run* option from within the IDE or we can load the source code from the executable file. The whole process is summarised in FIG-3.1.

FIG-3.1

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 3.1**
>
> 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?

Things are slightly more complicated when it comes to AGK2 BASIC.

For a start, an AGK2 BASIC program consists of several files and hence is referred to as a project rather than a program, with a new folder being created automatically for each new project.

But the real problem is that AGK programs are designed to run on a variety of devices (PC, MAC, Android tablet or phone, Apple tablet, or phone and the Ouya games console). Unfortunately, one device may use a different machine code from the next device. As a consequence, a binary pattern that means 'add' on one machine could quite possibly mean 'subtract' on another.

To get round this, AGK compiles our source program into something known as **bytecode**. Bytecode is the machine code for a computer that doesn't actually exist! Our nonexistent computer is known as a **virtual computer**.
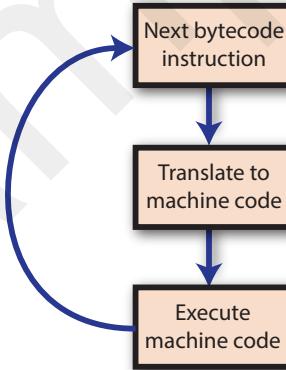
When we try to execute a program which has been translated into bytecode, a second program is loaded which emulates the virtual computer. In effect, this second program translates each bytecode statement (just before it is about to be executed) into actual machine code for the specific device on which the bytecode is being run.

This approach (which is also used by the Java programming language) is what allows our AGK program to run on so many devices, but at the cost of taking slightly longer to execute each statement because of the need to translate it from bytecode to true machine code.

The process is visualised in FIG-3.2.

FIG-3.2

Executing a
Bytecode Program

Functions

If you did even a little mathematics at school, you should have come across the idea of functions. Perhaps the most obvious being trigonometric functions such as sine, cosine, and tangent. How often did you have to suffer lines such as

    y = sin(45)

In the above line:

sin   is the name of the function.

45   is the **argument** or **parameter** value being passed to the function

The function itself performs a well-defined task. In the case of the *sin* function, it determines the sine of the angle given in the argument. The function "returns" the value it calculates.

All this is even more obvious if we are using a calculator. Type in the value 45, press the **sin** button and the value returned by the function, 0.707 (the sine of 45°), appears on the screen.

Functions in Programming

Functions play a pivotal role in computer programming. All large programs are split into a number of functions. Unlike the *sin* function above, a programming function can be designed to perform whatever task the programmer requires. It may be something as simple as clearing the computer screen or as complicated as calculating the interest due on a loan.

In general, functions have a name; take zero, one, or more parameters; and return a single result.

AGK comes with a large set of built-in functions. Some of these perform mathematical operations such as `sin()` and `cos()`, others are designed to set screen colour, text font, manipulate sprites or handle 3D objects.

A function is "called" by specifying its name, supplying a value for the argument, and making use of the value returned.

For example, the AGK2 BASIC code

```
y = Sin(0.7854)
```

calls the `Sin` standard function, supplying it with the value *0.7854* (angles are given in radians) and the value returned by the function is stored in a variable called *y*.

When referring to a function in this book, you'll usually see the function name followed by a set of parentheses as in `Sin()`. The parentheses are added simply to emphasize that the term refers to a function and not a variable name.

Starting AGK

Introduction

AGK is an Integrated Development Environment (IDE) software package designed to create 2D and 3D games that can then be run on various hardware devices.

AGK allows programs to be written in either BASIC or C++. This book covers only the BASIC language aspect of AGK.
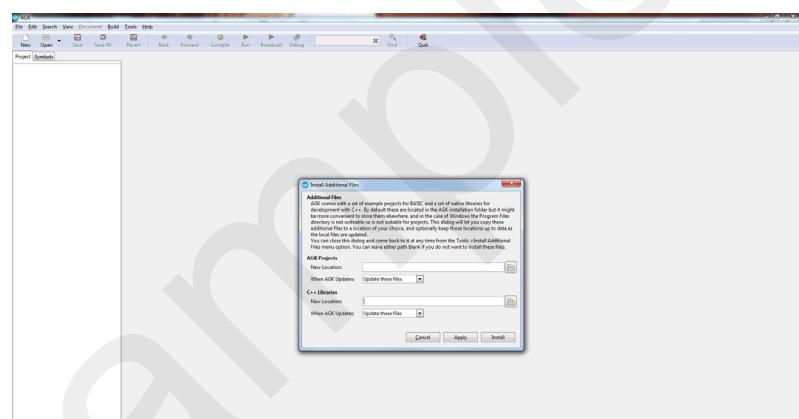
AGK was created by Lee Bamber, CEO of The Game Creators Ltd and was derived from his earlier creation, DarkBASIC which is a programming language designed to develop games for the PC platform only. Continued development of AGK is under the control of Paul Johnston and the IDE is based on the Geany text editor.

Starting Up AGK

Once you've installed AGK version 2, running the package will present you with the start-up screen shown in FIG-3.3.

FIG-3.3

The AGK Startup Screen

The dialog box at the centre of the screen may be ignored for the moment, so all that is required is to press the **Cancel** button.

To start your first project, follow the steps shown in FIG-3.4.

FIG-3.4

Creating a Project
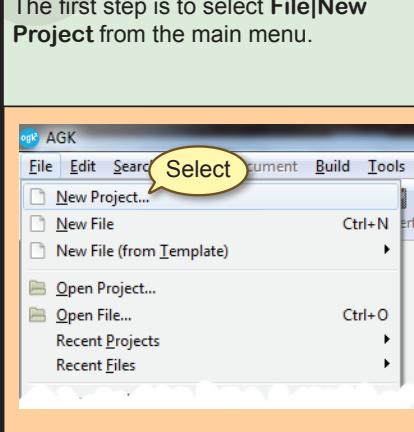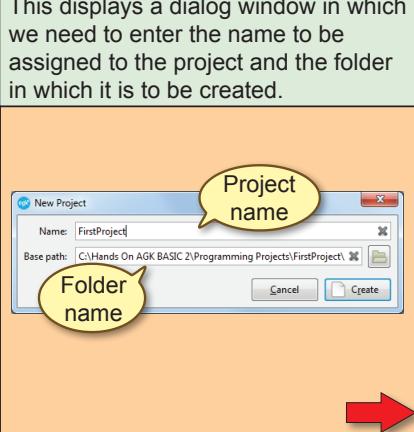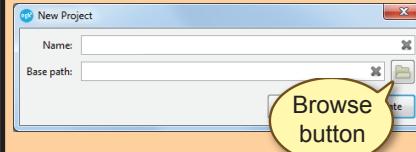
The first step is to select **File|New Project** from the main menu.	This displays a dialog window in which we need to enter the name to be assigned to the project and the folder in which it is to be created.

FIG-3.4
(continued)

Creating a Project
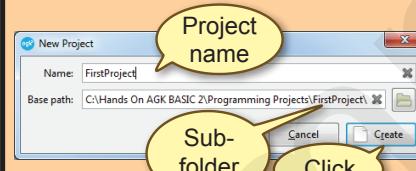
It's unlikely that the default folder will be where we want to store your new project, so we need to click on the browse button.	In the new dialog box we can select the folder we want to use.
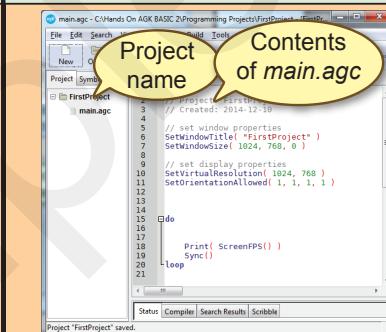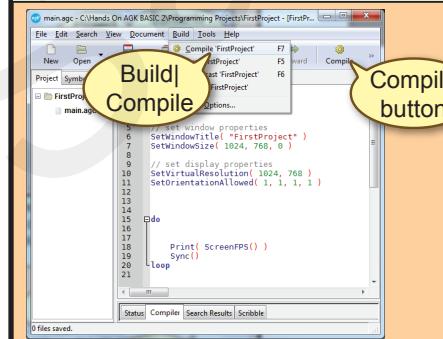
With the project name entered and the folder selected, press the **Create** button. The IDE will automatically create a subfolder to hold the project.	The created project is now listed in the **Project** page on the left along with the *main.agc* file which is part of the project. The contents of *main.agc* are displayed.
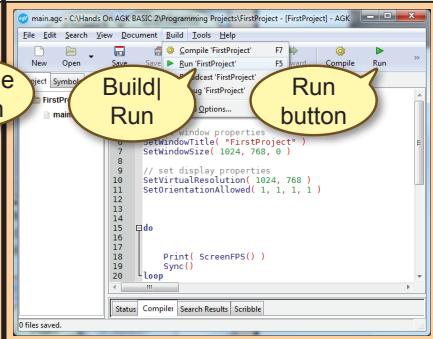
The default code in *main.agc* is a valid program. To convert it to bytecode we can press the **Compile** button, select **Build|Compile** or press **F7**.	The program is now ready to run. This is done by clicking on the **Run** button, selecting **Build|Run** or pressing F5.
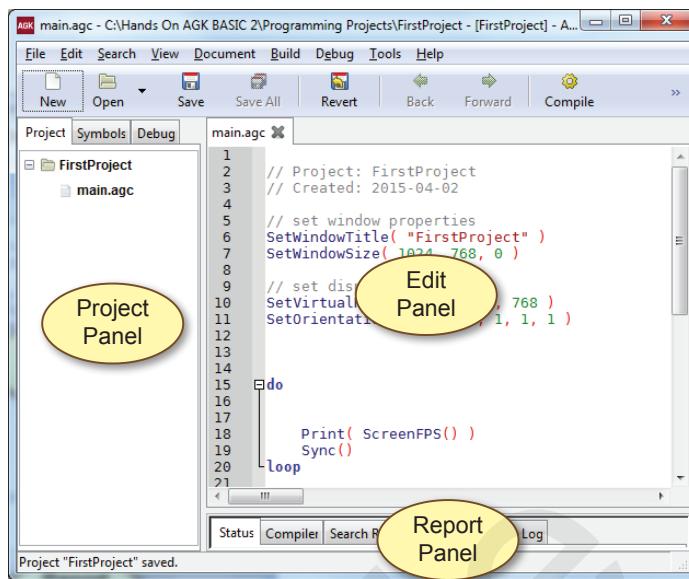
The file **main.agc**, with its default code, is created automatically with every project. We'll modify this code when creating our own programs.

When a project is created, the IDE displays three main areas (see FIG-3.5).

FIG-3.5

The IDE Layout

The Edit Panel	There can be many tabbed pages in this area, each showing the contents of a different file. Initially, the *main.agc* file for the new project is displayed.
The Project Panel	There are three tabbed pages in this panel. The first, labelled *Project*, displays the projects currently open and the files within those projects. The second page, labelled *Symbols*, contains details of various elements within the file currently being displayed in the *Edit Panel*. The third panel labelled *Debug* is used when attempting to find errors in a program.
The Report Panel	This contains five tabbed pages. The *Status* page displays information on the actions taken by the IDE. The *Compiler* page details the results of a compilation including a list of any syntax errors. The *Search Results* page displays the results produced by any searches within the currently displayed text file. The *Scribble* page allows us to make notes while we are working on a project. The *Debug Log* page displays messages created when running a program in debug mode.

All three panels can be resized.

Activity 3.2

Before you start up AGK, create a main folder called *HandsOnAGK2* on your disk drive. We'll use this as the main folder for all the AGK assets used throughout this book.

Create a subfolder off *HandsOnAGK2* called *Programming Projects*.

Load AGK and select the *Programming Projects* folder then create, compile, and run your first project (named *FirstProject*) exactly as described in FIG-3.4.
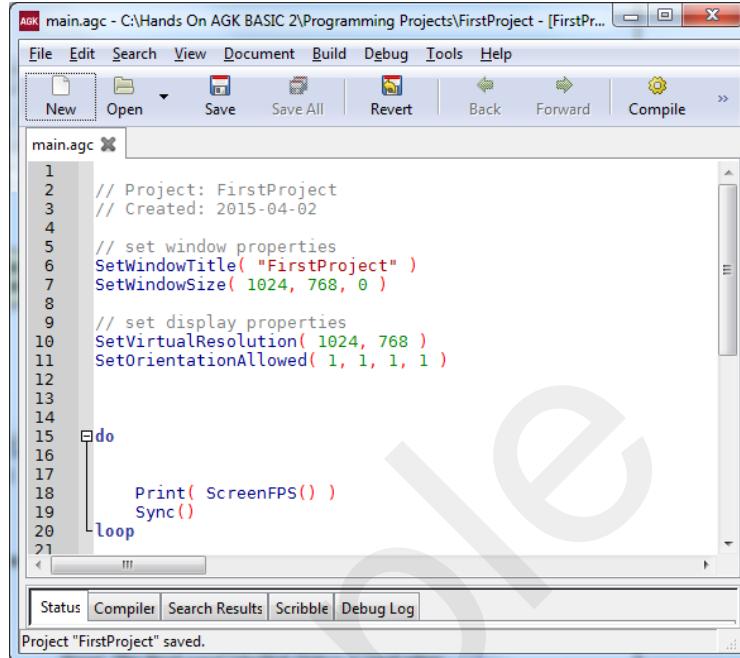
Close the app window after it has run for a few seconds.

The Program Code

FIG-3.6 shows the code in *main.agc* that was automatically generated for us.

FIG-3.6

The Generated
Code

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 program.

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

```
// Project: FirstProject
// Created: 2014-12-10
```

Text appearing after double forward slashes are treated as comments by the compiler and are ignored. In fact, the lines are there for the benefits of us humans reminding us of the project's name and the date on which it was created. Notice that the date is in the format year - month - day.

The next line is blank. Like comments, blank lines are ignored but are there to help with the visual appearance of the code, separating various sections of the program.

```
// set window properties
```

Another comment line. This time the comment tells us in plain English what the next lines of code are meant to achieve.

```
SetWindowTitle( "FirstProject" )
SetWindowSize( 1024, 768, 0 )
```

The first of these lines calls an AGK function which sets the text that will appear in the window's title area when the program is run. Notice that text values are enclosed

in quotes. Single or double quotes are allowed.
The second line calls another function which sets the size of that window (1024 pixels wide by 768 pixels high). The third value in the parentheses (0), states that the app should run in a standard window rather than in a borderless, full screen mode.

```
// set display properties
SetVirtualResolution( 1024, 768 )
SetOrientationAllowed( 1, 1, 1, 1 )
```

The first function call specifies the assumed size of the window in pixels (width then height). This need not be the same values as those used in the call to `SetWindowSize()` but we'll have more to say on this topic later.
The second function call is really for tablets and phones and specifies that the layout produced by the program will rotate along with the device on which it is being displayed.

```
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( ScreenFPS() )
```

The `Print()` statement is used to display information in the app window. The information itself is specified within parentheses. In this case what is being displayed is the frame rate of your hardware - this tells us how many times per second a new image can be displayed on the screen (**F**rames **P**er **S**econd).

```
Sync()
```

The `Sync()` function updates 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.

Activity 3.3

In *FirstProject*, modify the line containing the `Print` statement so that it reads

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

Compile and run the program. What is displayed this time?

To save your program, select **File|Save** from the main menu bar.

Running Your App on a Tablet or Smartphone

Producing a true app for your smartphone or tablet will be covered in a later chapter, but you can, nevertheless, watch your app run on such a device before the final version is produced. On an Android or iOS device, first download the app **AGK Player 2** from the app store used by your device. You will find *AGK Player 2* in *Google Play* and *Apple Store.*

Before running an app on the target device, make sure it and your desktop are both connected to your local WiFi network. This will allow the desktop to broadcast the app to the target device. Now load the source code of the app into the AGK IDE on your desktop and start the *AGK Player 2* app running on the mobile device. Next, press AGK's **Broadcast** button. The *AGK Player 2* app will detect your program being broadcast, then download and run it on your device. The steps involved are shown in FIG-3.7.

Broadcast

FIG-3.7

Installing and Using AGK Player 2

On your Android device, select **Play Store** (Google Play). On your iOS device, select **App Store**.	When connected to the store, search for *AGK Player 2*.
Click on the AGK2 app icon and click on **Install**.	You will now see the AGK Player 2 icon on your device.
Touch the app icon to start the player running. Your device will now display the screen shown below.	On your PC, with the program you want to run already compiled, press the **Broadcast** button in AGK's IDE. Your app will then appear on your device.

When you want to terminate your app running within *AGK Player 2*, press your finger at a position near the top of the screen for 5 seconds to return to the *AGK Player 2* screen.

Activity 3.4

Make sure you have the *AGK Player2* app installed and running on your device.

With the latest version of *FirstProject* you created in Activity 3.3 showing on the AGK IDE, press the **Broadcast** button. Check that the program is now showing on your device.

Press the top of the screen for 5 seconds to exit your own app and return to *AGK Player2*.

First Statements in AGK2 BASIC

Introduction

Learning to program in AGK2 BASIC is very simple compared to other languages such as C++ or Java. Unlike most other programming languages, it has no rigid structure that the program itself must adhere to.

Now we need to start looking at the formal statements allowed in AGK2 BASIC and see how they can be used in a program.

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 AGK2 BASIC there are five alternative ways to add comments:

Add the keyword `rem`. The remainder of the line becomes a comment (see FIG-3.8).

FIG-3.8

rem

Add an apostrophe character (you'll find this on the top left key, just next to the *1* key on a PC). Again the remainder of the line is treated as a comment (see FIG-3.9).

FIG-3.9

Apostrophe
Comments

Add two forward slashes to make the remainder of the line a comment (see FIG-3.10).
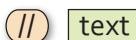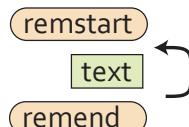
FIG-3.10

// Comments

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-3.11).

FIG-3.11

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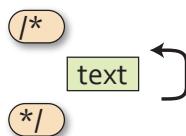
This last diagram introduces another symbol - a looping arrowed line. This is used to indicate a section of the structure that may be repeated if required. In the diagram above it is used to signify that any number of comment lines can be placed between the `remstart` and `remend` keywords. For example,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

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

The final comments option is another used to create a comments block and is an alternative to `remstart` and `remend` (see FIG-3.12).

FIG-3.12

/* */ Comment
Block

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-3.13.

FIG-3.13

Print()

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

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

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

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

```
Print("Hello world")
Print('Help!')
Print(12)
Print(0)
Print(-34.6)
```

while the following are not:

```
Print 36          (parentheses are missing)
Print(Goodbye)    (no quotes enclosing Goodbye)
```

Activity 3.5

Which of the following are NOT valid `Print()` statements:
 a)   `Print("-9.7")`
 b)   `Print(0.0)`
 c)   `Print(23, 51)`

Spaces

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

```
Print    (    123    )
```

since each token can easily be identified, but

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

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

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

```
if x = 3
```

then writing

```
ifx=3
```

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

```
Print(123)
```

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

Multiple Output

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

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

are included in a program, they will create the output

```
Hello
Goodbye
```

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

> **Activity 3.6**
>
> Modify *FirstProject* so that the main code now reads
> ```
> do
>         Print("First line")
>         Print("Second line")
>         Sync()
> loop
> ```
> Compile and run the program.

PrintC()

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

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

would display

AB

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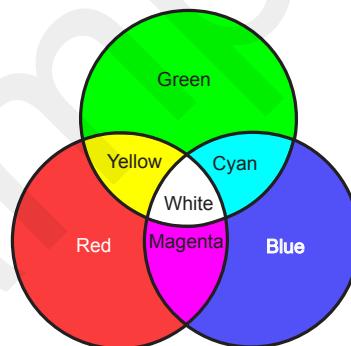
Other statements allow us to make various changes to how the information appearing
on our screen is presented. We can change the text colour, size, transparency and even
the space between the characters.

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 display screen are derived from the three primary colours
red, green and blue. By varying the brightness of each of these three colours we can
achieve almost any colour or shade the eye is capable of seeing. For example, mixing
just red and green gives us yellow; blue and green gives us a colour called cyan, and
blue and red gives magenta (see FIG-3.14).

FIG-3.14

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 opacity of the text.

FIG-3.15

SetPrintColor()

The statement's format is shown in FIG-3.15.

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:

red	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. Default value: 255
green	is an integer value (0 to 255) giving the strength of the green component. Default value: 255
blue	is an integer value (0 to 255) giving the strength of the blue component. Default value: 255
opac	is an integer value (0 to 255) giving the opacity of the text. 0 – invisible, 255 – fully opaque. Default value: 255

Text output using the default values is white opaque.

Since the opacity value is optional and therefore can be omitted (in which case opacity 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)       //*** sets text to black
SetPrintColor(255,255,255) //*** sets text to white (default)
SetPrintColor(255,0,0)     //*** sets text to red
```

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

> **Activity 3.8**
>
> In *FirstProject*, 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 3.9**
>
> In *FirstProject*, reposition your `SetPrintColor()` statement, placing it on the line above `do`.
>
> Compile and run the program again. Does this change the text colour?

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 3.10

Modify the `SetPrintColor()` statement in *FirstProject*, adding 126 as the transparency value.

Run the program and see what effect the changes have made to the output. Try other transparency values to see their effect.

SetPrintSize()

The `SetPrintSize()` statement (see FIG-3.16) sets the size of the text displayed by a `Print()` or `PrintC()` statement.

where:

size	is a real number setting the size of the characters. The default value for characters is about 3.3%.

Activity 3.11

Add the line
```
SetPrintSize(40)
```
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-3.17) adjusts the spacing between the characters shown on the screen.

where:

gap	is a real number giving the gap between the characters. The default is zero. Larger values widen the gap; negative values cause the gap to decrease and even to make letters overlap.

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-3.18.

SetClearColor ( ) red , green , blue)

where:

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

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

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

ClearScreen()

The `SetClearColor()` statement only works when followed by a `Sync()` or a `ClearScreen()` statement which has the same effect. The format for the `ClearScreen()` statement is given in FIG-3.19.

ClearScreen ( )

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

```
SetClearColor(255,255,0)
ClearScreen()
```

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

Positioning the Print() Statements

We have placed the various statements affecting the colour, size and spacing of our text before the `do...loop` structure on the basis that these commands need only be performed once. So you may be tempted to think that surely we can do the same thing

with the `Print()` and `Sync()` statements since the displayed text remains unchanged throughout the running of the program. Let's see what happens when we try this.

Activity 3.14

Move the `PrintC()` and `Sync()` statements in *FirstProject* 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 produced, for a simple program such as this, moving the statements has had no effect on the output produced. We are left with an empty `do...loop` which makes sure that the program does not terminate before we click the app window's **Close** button.

From what was said about creating efficient code it might seem like a good idea to move the `Print()` and `Sync()` statements outside the loop. However, the `Sync()` statement does more than just update the screen display (more on this later) and with it placed outside the main loop we may run into various problems, so make sure you have at least one call to `Sync()` in the program's `do...loop` structure.

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-3.20.

FIG-3.20

Message()

where

text   is a string containing the message to be displayed.

For example, the line

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

produces the output shown in FIG-3.21 when run on a PC.

FIG-3.21

A Typical Message Window

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

Summary

■ Programs are written using a programming language.

■ Programming language code must be translated into machine code before the program can be executed by the computer.

- The program code is known as the source code; the machine code as the object code.

- The object code created by the AGK compiler is a form of bytecode.

- The saved object code is known as an executable file.

- When an AGK program is run the bytecode is translated into the machine code for that hardware.

- Each line of a program must conform to the rules of syntax.

- An error in how a line is written is known as a syntax error.

- AGK programs can be written in BASIC or C++.

- The collection of files created when writing an AGK app is known as a project.

- The main file in an AGK project is *main.agc* which contains the program code.

- The AGK development package is an Integrated Development Environment. This allows edit, compiling and testing to be performed from within the same program.

- To download an app to your digital device, the *AGK Player 2* app must be installed and running on that device and the app broadcast from the AGK IDE.

- When an app is being tested on a desktop, it creates an app window.

- Comments can be added to your code using `//`, `rem`, `` ` ``, `remstart...remend` or `/*...*/`.

- Comments help us understand the purpose of a piece of code but are ignored by the compiler.

- Use `Print()` to display information on the screen.

- Use `PrintC()` to display information without moving to a new line afterwards.

- Use `SetPrintColor()` to set the colour used when displaying text.

- Use `SetPrintSize()` to set the size of future text output.

- Use `SetPrintSpacing()` to set the spacing between characters in future text output.

- Use `Message()` to display a message in a separate window.

- Use `SetClearColor()` to set a background colour for the app screen.

- Use `ClearScreen()` to clear the PC's app window or the device's screen.

App Window Properties

Running Apps on a Desktop

When an app is designed to run exclusively on a desktop or laptop, then we have a fairly simple job to do in defining the window size and the window's title.

SetWindowSize()

The `SetWindowSize()` command sets the width and height of an app's window when run on a desktop. The command is ignored if the app is run on a mobile device. The format of the statement is given in FIG-3.22.

FIG-3.22

SetWindowSize()

SetWindowSize ( ) width , height , opt)

where

width	is an integer value giving the width of the window in pixels.
height	is an integer value giving the height of the window in pixels.
opt	is an integer value (0 or 1) used to create a standard window (0) or a borderless full-screen window (1). If option 1 is used, the width and height values are ignored.

A typical statement would be

```
SetWindowSize(800,600,0)
```

which would create a window 800 pixels wide by 600 pixels high.

SetWindowTitle()

We can set the title that appears at the top of the window using the `SetWindowTitle()` statement (see FIG-3.23).

FIG-3.23

SetWindowTitle()

SetWindowTitle ( ) text)

where

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

A typical statement would be:

```
SetWindowTitle("Jigsaw Game")
```
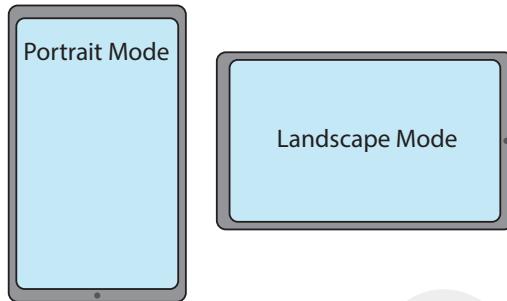
> **Activity 3.15**
>
> Modify *FirstProject* so that the window's dimensions are 900 by 750 pixels and contains the title *"My First AGK Project"*. Run the program and check that the title appears in the window.

Screen Orientation

When we create an app for a mobile device, we may expect it to be run with the screen of the target device oriented in a specific way. If we want the longest side to vertical then the screen is said to be in **portrait mode**; with the longest side horizontal, the screen is in **landscape mode** (see FIG-3.24).

FIG-3.24
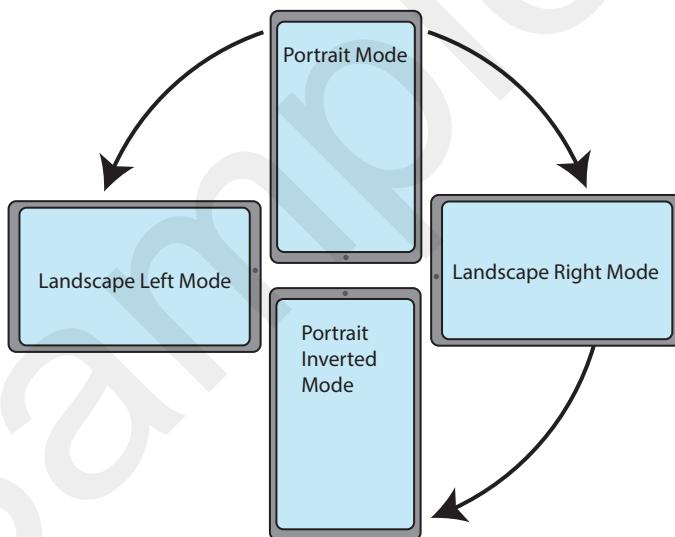
Basic Screen Orientations

Of course, with mobile devices, the user may orient the screen in any one of four ways (see FIG-3.25).

FIG-3.25

All Screen Orientations

In designing an app, we can decide if the image on the screen is going to rotate when the screen is moved from one orientation to another, or if it is going to remain unchanged. We can even specify that it should change for some orientations and not for others. This is achieved using the `SetOrientationAllowed()` function (see FIG-3.26).

FIG-3.26

SetOrientationAllowed()

where:

port   {is 0 or 1} 1 allows portrait orientation, 0 does not allow this orientation.

invport   {0 or 1} 1 allows inverted portrait orientation, 0 does not allow this orientation.

landleft	{0 or 1} 1 allows landscape left orientation, 0 does not allow this orientation.
landright	{0 or 1} 1 allows landscape right orientation, 0 does not allow this orientation.

If an app initially filled the portrait mode with the image shown in FIG-3.27.

FIG-3.27

Image in Portrait Mode

Portrait Mode
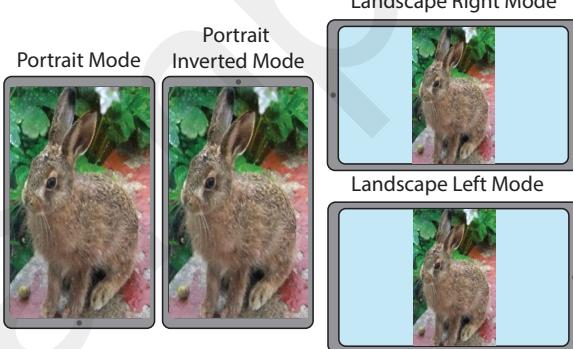
and made use of the line

```
SetOrientationAllowed(1,1,1,1)
```

then it would create the screens shown in FIG-3.28 as it was rotated into the other orientations.

FIG-3.28

Displays Produced when Each Orientation Allowed

Portrait Mode

Portrait Inverted Mode

Landscape Right Mode

Landscape Left Mode
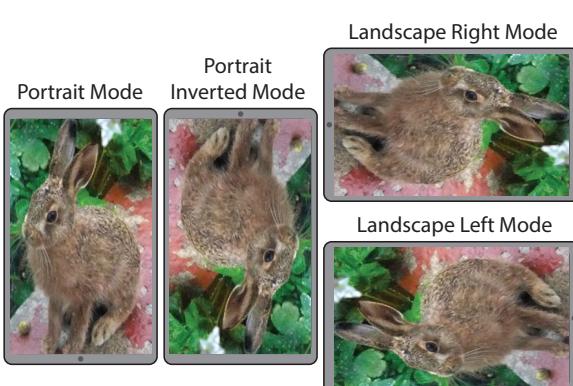
On the other hand, using the line

```
SetOrietationAllowed(1,0,0,0)
```

would produce the results shown in FIG-3.29.

FIG-3.29

Displays Produced when Only Portrait Orientation Allowed

Portrait Mode

Portrait Inverted Mode

Landscape Right Mode

Landscape Left Mode

If we do decide to allow the screen to change with different orientations, we may have to modify the code executed when producing the layout in order to achieve exactly the effects we require.
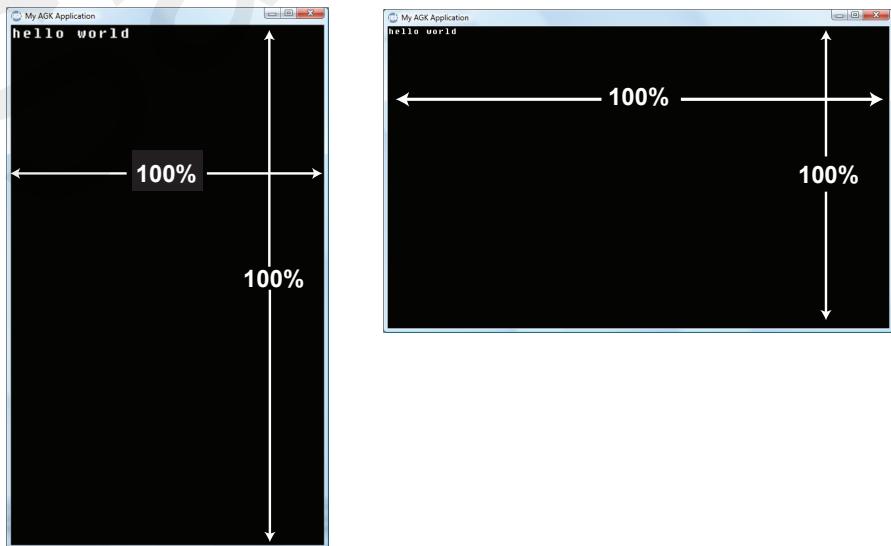
Of course, if your app is designed to run on a desktop or standard laptop, then the orientation of the app will not change.

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-3.30).

FIG-3.30

The Screen's Percentage
Measurement System

When you want to position an item on the screen it is done using percentage

measurements. This means that the position (50,50) represents the middle of the app window irrespective of the window's actual dimensions.

The percentage system is ideal in many ways, since it allows us to worry less about the physical resolution of the devices on which our app runs. For instance, if we give an element of the screen a width of 50%, then we know it will take up half the width of the screen, no matter what the actual resolution of that screen may be.

On the other hand, we have to realise that the actual size of an element may change when shown on different devices. Let's say a screen has a resolution of 768 pixels wide by 1024 pixels high, then text which is defined to have a height of 2% will in reality be about 20 pixels high, but on a screen with a resolution of 1536 by 2048, it will be 40 pixels high.

The other thing of importance is the pixel density. That is to say, the pixels per inch (or per centimetre). The first two iPad models had a 768 by 1024 pixel screen, while all later models (so far) have had a resolution of 1536 by 2048, but all models have had exactly the same physical screen size (9.7 inches diagonal) so although a text element defined to be 2% high uses 20 pixels in the earlier models and 40 in the later models, the physical size of the text on the screen would be the same in all models (though the text should look better on the higher resolution screen).

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 this effect in the *"Hello world"* text visible in FIG-3.28 shown earlier.

Activity 3.17

Let's suppose we are going to run our completed app on three different devices which have the following resolutions when in portrait mode :

    a) 640×1136
    b) 800×1280
    c) 1536×2048

To the nearest pixel, how many pixels tall would text defined with a height of 2% be on each device?

All programs in this book use this default percentage system.

SetVirtualResolution()

If you would rather work with a resolution based on pixels rather than percentages, have your program execute the `SetVirtualResolution()` function when it starts up. The statement's format is shown in FIG-3.31.

FIG-3.31

SetVirtualResolution()

where:

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

If you were writing a portrait mode app for the iPhone 2, you would set the resolution to 320×480 using the line:

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

However, when you transfer the app to another device, the app will adjust to fit that device's screen. For example, if you run your 320x480 app on a newer iPhone 4 with its 640x960 resolution, your AGK app will automatically expand to fill the full screen. Of course, this means that although your code might state that text is to be, say, 10 pixels high, it would in fact be 20 pixels high when run on the higher resolution iPhone 4.

This is why the term **virtual resolution** is used; the resolution defined in the program may be different from the actual resolution used when the app is running on a device.

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

Activity 3.18

An app defines the screen's virtual resolution to be 640x960. How many actual pixels would 1 virtual pixel represent when run on a device with the following screen resolution:

    a) 320×480
    b) 640×960
    c) 1280×1920

When we are developing our app on a desktop, the app window's physical size is not affected by the virtual resolution settings. Use `SetWindowSize()` to set the actual dimensions of the window.

SetDisplayAspect()

Every program must define its aspect ratio. That is the ratio of the screen's (or window's) width to its height. This is done using the `SetDisplayAspect()` function (see FIG-3.32).

FIG-3.32

SetDisplayAspect()

where:

ratio	is a floating point number giving the width to height ratio. For example, all iPhone and iPad devices have an aspect ratio of 4.0/3.0 (1.3333) when in landscape mode. A value of -1 uses the whole screen's 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 by 3

(4/3) programme on your widescreen TV).

If this statement is omitted from a program using the percentage system, other commands may not operate correctly.

Handling Different Display Aspects

A problem arises when the device on which your app is running has a different aspect ration (width / height) than that specified in the `SetVirtualResolution()` statement. Expanding the app's resolution from 320x480 to 640x960 isn't a problem because both have an aspect ratio of 3/4. But if we were to try and run the same app on an original Asus EEE Transformer which has a resolution of 800x1280 (an aspect ratio of 5/8) then we have a problem. Expanding the app to fill a 5/8 screen would cause distortion of any images being displayed (circles would become ovals!).

AGK handles this change of resolution by creating as large a 3/4 ratio image as possible and adding a border to the remainder of the screen.

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-3.33).

$$\boxed{\text{SetBorderColor}} \, \boxed{(\,)} \, \boxed{\text{red}} \, \boxed{,} \, \boxed{\text{green}} \, \boxed{,} \, \boxed{\text{blue}} \, \boxed{(\,)}$$

where:

red   is an integer value (0 to 255) giving the intensity of the red component of the border colour to be used. 0: no red; 255: full red.

green   is an integer value (0 to 255) giving the intensity of the green component of the border colour. 0: no green; 255: full green.

blue   is an integer value (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)
```

SetWindowAllowResize()

This function determines if the user is allowed to resize the app's window. The function has the format shown in FIG-3.34.

$$\boxed{\text{SetWindowAllowResize}} \, \boxed{(\,)} \, \boxed{\text{opt}} \, \boxed{(\,)}$$

where

opt   is an integer value (0 or 1) which disables resizing (0) or allows it (1).

MinimizeApp()

This function minimises the app. On a desktop, the app will minimise but continue to execute. Under Android, the app will pause. Under iOS, the command has no effect. The function has the format shown in FIG-3.35.

$$\text{MinimizeApp} \; (\;)$$

SetWindowPosition()

If we want to position set the initial position of the app window when running on a desktop, we can use `SetWindowPosition()` (see FIG-3.36).

$$\text{SetWindowPosition} \; (\; \boxed{x} \; , \; \boxed{y} \;)$$

where:

x,y	are integer values giving the coordinates of the top-left corner of the window's new position.

Summary

- Use `SetWindowSize()` to set the dimensions of the app window (or use full-screen) when using a desktop machine.

- Any `SetWindowSize()` statement is ignored when an app is run on a mobile device.

- Use `SetWindowTitle()` to set the title displayed by a windowed app on a desktop.

- Use `SetOrientationAllowed()` to set which orientations the app screen will rotate to when run on a mobile device.

- By default, AGK uses a percentage coordinate system within the app window.

- Use `SetVirtualResolution()` to use a virtual pixel coordinate system.

- Use `SetDisplayAspect()` to set the width to height ratio of the screen/window.

- If a call to `SetDisplayAspect()` is omitted from a program, other commands may not function correctly.

- Use `SetBorderColor()` to specify a colour for any part of the physical screen not included in the app's output area.

- Use `SetWindowAllowResize()` to allow/disallow window resizing.

- Use `MinimizeApp()` to minimise an app.

- Use `SetWindowPosition()` to set the initial position of the window when creating a desktop app.

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 while other resources such as images and sound files are loaded into memory. Typically a splash screen will contain an image giving the flavour of the game play that is about to follow as well as the name of the game and the publishing company.

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

If you load Windows Explorer and have a look in the folder created by AGK to hold the files belonging to your project (*HandsOnAGK2/Programming Projects/FirstProject*), you should see contents similar to that shown in FIG-3.36.

FIG-3.36

AGK Project's Files

The splash screen graphic file must be placed in the project's main folder. The file must be in **PNG** format and be called *AGKSplash.png*. No other name is acceptable. The image is best set to the same size as the window dimensions (in our case, 1024 x 768). An example of a splash screen is shown in FIG-3.37.

FIG-3.37

A Splash Screen

Activity 3.19

Load the file *AGKSplash.png* from the resources you downloaded for this chapter and place it in your *FirstProject* folder.

In AGK, change the dimensions of the app window to 480 by 640 (the size of the image) then recompile your program and run it.

Does the splash screen appear when the app window first opens?

As you can see from the results of Activity 3.19, the splash screen is not ideal. In fact, things are even worse if you run the app through *AGK Player2* on your mobile device – there will be no sign of the splash screen at all!

In a later chapter we'll see a way to code a better behaved splash screen.
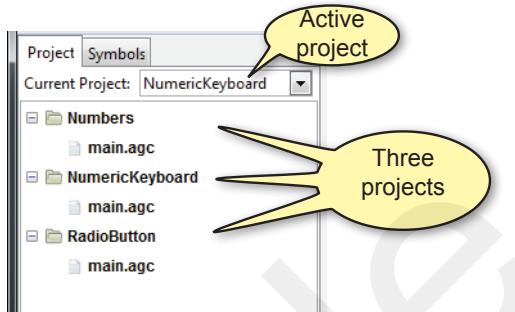
Starting a New Project

Should you want to create a second project or open another, existing project, you can do so from the main menu (**File|New Project** or **File|Open Project**).

Every open project appears in the *Projects Panel* on the left side of the window. Which of the multiple projects is currently active is shown in the combobox at the top right of the panel (see FIG-3.38).

FIG-3.38
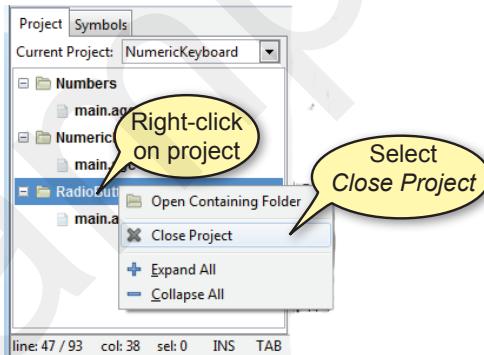
Multiple Projects

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. To do this, right-click on the project to be closed and select **Close Project** from the pop-up menu (see FIG-3.39).

FIG-3.39

Closing a Project

Solutions

Activity 3.1

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

Activity 3.2

No solution required.

Activity 3.3

Your code for the `do..loop` should now read

```
do
        Print("Hello world")
        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** to save the updated code.

Activity 3.4

If you are downloading AGKPlayer from Google Play:
        Run the Google Play app
        Search for AGK Player2
        Install the app

Activity 3.5

a) Valid. Any characters can be enclosed in quotes - including numeric ones.

b) Valid. A floating-point number.

c) Invalid. Only a single value can be displayed.

Activity 3.6

The output should be:

```
First line
Second line
```

Activity 3.7

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 3.8

Program code (your colour values will be different):

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

Activity 3.9

Program code (your colour values will be different):

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

Activity 3.10

Program code (your colour values will be different):

```
//*** Use yellow, translucent text ***
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 3.11

Program code (your colour values will be different):

```
//*** Use yellow, translucent text ***
SetPrintColor(255,255,0,126) rem *** yellow ***
//*** Set text size to 40 ***
SetPrintSize(40)
do
    PrintC("First line")
    PrintC("Second line")
    Sync()
loop
```

The text will appear larger but somewhat blurred.

Activity 3.12

Program code (your colour values will be different):

```
//*** Use yellow translucent size 40, 5.5 spaced
text ***
SetPrintColor(255,255,0,126) rem *** yellow ***
SetPrintSize(40)
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(-3.5)
```
The characters will now bunch together.

Activity 3.13

Program code:

```
//*** Use a yellow background ***
SetClearColor(255,0,0)
ClearScreen()
//*** Use yellow translucent size 40, -3.5 spaced
text ***
SetPrintColor(255,255,0,126)
SetPrintSize(40)
SetPrintSpacing(-3.5)
do
    PrintC("First line")
```

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

Activity 3.14

Program code:

```
// Project: FirstProject
// Created: 2014-12-29

// set window properties
SetWindowTitle( "FirstProject" )
SetWindowSize( 1024, 768, 0 )

// set display properties
SetVirtualResolution( 1024, 768 )
SetOrientationAllowed( 1, 1, 1, 1 )

//*** Create a red background ***
SetClearColor(255,0,0)
ClearScreen()

//*** Use yellow, size 40, -3.5 spaced text ***
SetPrintColor(255,255,0,126)
SetPrintSize(40)
SetPrintSpacing(-3.5)

//*** Display message ***
PrintC("First line")
PrintC("Second line")
Sync()
do
loop
```

The output remains unchanged.

Activity 3.15

Change the lines

```
SetWindowSize(1024,768,0)
SetWindowTitle("FirstProject")
```

to read

```
SetWindowSize(900,750,0)
SetWindowTitle("My First AGK Project")
```

Activity 3.16

With the orientation set to allow Portrait and Right Landscape the screen would display:

> Portrait : A
> Inverse Portrait : B
> Right Landscape: D
> Left Landscape : F

Activity 3.17

> a) 23 pixels (2*1136/100)
> b) 26 pixels (2*1280/100)
> c) 41 pixels (2*20148/100)

Activity 3.18

> a) 0.5 pixels horizontally; 0.5 pixels vertically
> b) 1 pixel horizontally; 1 pixel vertically
> c) 2 pixels horizontally; 2 pixels vertically

Activity 3.19

Although the splash screen appears as expected, the window isn't correctly sized at that point, so the image is not an exact fit for the window. And when the window does change to the size specified in the code, the splash screen is no longer visible!

4

Data

In this Chapter:

- ❏ Constants
- ❏ Variable Types
- ❏ Naming Variables
- ❏ Declaring Variables
- ❏ Named Constants
- ❏ The Assignment Statement
- ❏ Arithmetic Operators
- ❏ Operator Priority
- ❏ **inc** and **dec** Statements
- ❏ The **Mod()** Function
- ❏ Timer Functions
- ❏ Random Functions
- ❏ User Input
- ❏ **#include**
- ❏ Testing Sequential Code

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 simple data can be grouped into three basic types:

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

✎ The variables that hold **real** values are often referred to as **float** or **floating-point** variables. This reflects the storage format used.

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

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

> **Activity 4.1**
>
> Identify the type of value for each of the following :
>
> a) -9   b) abc   c) 18   d) 12.8
> e) ?   f) 0   g) -4.0   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** or **literal**. 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 4.2**
>
> What type of constants are the following:
>
> a) -12   b) Elizabeth   c) 4.14   d) 27.0

Variables

To store data in AGK BASIC we need to make use of a **variable**. A variable is, in effect, reserved space within the computer's memory where a single value can be stored.

Every variable in a program is assigned a unique name and can store only a single value at any moment in time.

When a variable is first created, the type of value it can store (integer, real or string) is specified implicitly or explicitly. No other type of value can be stored in that variable. For instance, a variable designed to store an integer value cannot store a string.

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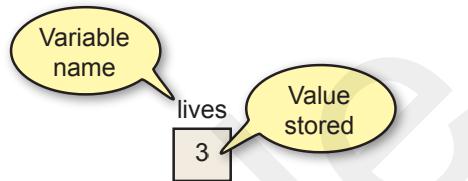
In AGK BASIC, variables are created automatically as soon as we mention them in our code. For example, let's assume we want to store the number of lives allocated to a game player in a variable called *lives*. To do this, we simply write the line:

```
lives = 3
```

This sets up a variable called *lives* in the computer's memory and stores the value 3 in that variable (see FIG-4.1)

This instruction is known as an **assignment statement** since we are assigning a value (3) to a variable (*lives*).
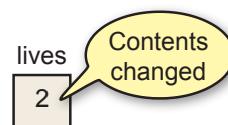
We are free to change the contents of a variable at any time by assigning it a different value. For example, later in the program, we can change the contents of *lives* with a line such as:

```
lives = 2
```

When we do this, any previous value stored in the variable will be removed and the new value stored in its place (see FIG-4.2).

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 three assignments are correct

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

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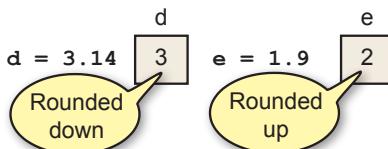
```
d = 3.14
e = 1.9
```

will cause problems since they attempt to store real constants in variables designed to hold integers.

AGK BASIC won't actually report an error if we 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-4.3). Fractions of 0.5 and above are rounded up, other values are rounded down.

FIG-4.3

d = 3.14   3 (Rounded down)   e = 1.9   2 (Rounded up)

Floating-Point Variables
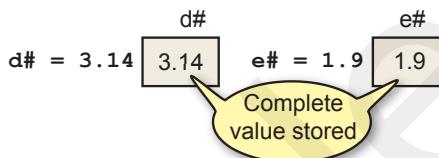
If we want to create a variable capable of storing a number with a decimal point, then we must end the variable name with the hash (#) symbol. For example, if we write

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

we have created variables named *d#* and *e#*, both capable of storing real values (see FIG-4.4).

FIG-4.4

Float Variables

d# = 3.14   3.14   e# = 1.9   1.9
Complete value stored

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

```
d# = 12
```

Although we may assign an integer to a float variable, the value will be stored in floating-point format. Therefore, after the statement above has been executed, *d#* will contain the binary equivalent of 12.0.

If any numeric value can be stored in a float variable, why bother with integer variables? Actually, we should always use integer values wherever possible because some hardware can be much faster at handling integer values than float ones. Also, floating-point numbers can be slightly inaccurate because of rounding errors within the machine (see Chapter 2). For example, the value 2.3 might be stored as the binary equivalent of 2.2999987. Another consideration is that a floating-point variable usually requires more space in the computer's memory than an integer one.

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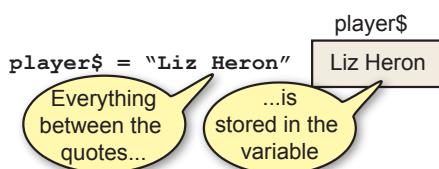
Finally, if we want to store a string value, we need to use a string variable. String variable names must end with a dollar (*$*) sign. The value to be stored must be enclosed in single or double quotes. We could create a string variable named *player$* and store the name *Liz Heron* in it using the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

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

The quotes are not stored in the variable (see FIG-4.5).

FIG-4.5

String Variables

player$ = "Liz Heron"   Liz Heron
Everything between the quotes...   ...is stored in the variable

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

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

Using Meaningful Names

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

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

a better set of statements would be

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

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

Naming Rules

AGK BASIC, like all other programming languages, demands that we follow a few rules when we make up a variable name. The rules for this language 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 float 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.

✎
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.

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, we 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*).

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

Declaring Variables

Many programming languages demand that we explicitly declare variables (stating their name and the type of value they are to hold) before using them. For example, in C++, an error would be generated if we were to write

```
no = 12;
```

before having declared *no* as an integer variable with the statement

```
int no;
```

Although AGK BASIC does not enforce variable declaration in the same way as C++, it nevertheless gives us the option to declare variables with code such as

```
lives           as integer
interest_rate   as float
name            as string
```

Note that when we use this approach we are no longer required to end float variable names with the # symbol, nor string variables with a *$* character.

#option_explicit

We can even tell the AGK BASIC compiler that all variables MUST be declared by adding the compiler directive

```
#option_explicit
```

at the start of a program.

With this directive in place, the line

```
lives = 3
```

would be invalid without the previous declaration:

```
lives as integer
```

We can actually combine these two statements, giving the variable *lives* a value at the moment it is declared:

```
lives as integer = 3
```

The format for explicitly declaring a variable is shown in FIG-4.6.

FIG-4.6

Declaring Variables

$$\boxed{\text{name}}\ \widehat{\text{as}}\ \boxed{\text{type}}\ \Big[\ \widehat{=}\ \boxed{\text{value}}\ \Big]$$

where:

name   is the name being given to the variable.

type   is the variable's type. At this point, we know only of types `integer`, `float` and `string`.

| | value | is the initial value to be assigned to the variable. |

But why would we want to declare variables explicitly? Well, the main reason is because of the mistakes we are going to make when writing a program! Somewhere along the line we are going to make a mistake something like this:

A line in our program reads

```
no1 = 12
```

and many lines later we intend to change this variable's value to 6 with the statement

```
nol = 6
```

Looks okay? It's not! In the first instance the last character in the variable name is the numeric digit 1 (one) but in the second line its a lowercase L.

AGK BASIC won't object to this the difference: it just assumes we are using two different variables!

But if we had added `#option_explicit` to our code, and declared *no1* (with a one) as an integer variable, the compiler would have reported the fact that *nol* (with a lowercase L) had not been declared, allowing us to spot our error instantly.

Named Constants

#constant

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 fixed value using the `#constant` statement. The format of the `#constant` statement is shown in FIG-4.7.

FIG-4.7

#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 named *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 literals or constants.

- There are three types of constants: integer, real and string.

- String constants are always enclosed in single or double quotes.

- The 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.

- Variables may be explicitly declared before they are used.

- When variables are declared, float variable names need not end with a # symbol and string variables need not end with *$*.

- Use #`option_explicit` if you wish the declaration of variables to be compulsory.

- Use `#constant` to create named constants.

- Traditionally, named constants have names in uppercase.

- Real and string named constants must not end with a # or *$* symbol.

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 statements 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-4.8.

| variable | = | value |

where:

variable	is the name of the variable being assigned a value.
value	is one of the following:
	a constant
	another variable
	an arithmetic expression

Examples of each type of value 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.

> **Activity 4.5**
>
> What are the minimum changes required to make the following statements operate correctly?
>
> a)   `desc = "tail"`   b)   `result = 12.34`

Copying Another Variable's Value

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

```
no1 = 12
```

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

```
no2 = no1
```

When the assignment is complete, both variables will contain the value 12. As before, we must make sure the two variables are of the same type, although the

contents of an integer variable may be copied to a float variable as in the line:

```
ans# = no1
```

Copying the contents of a float 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 4.6

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 5 possible operators that may be used when performing a calculation. These are shown in FIG-4.9.

FIG-4.9

Arithmetic Operators

Operator	Function	Example
+	addition	no1 = no2 + 5
-	subtraction	no1 = no2 - 9
*	multiplication	ans = no1 * no2
/	division	r1# = n01/ 2.0
^	power	ans = 2^3

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

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

and then contains the line

```
total = no1 - no2
```

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

```
product = no1 * no2
```

stores the value 36 in the variable *product*.

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

```
addresses = 2^24
```

Activity 4.7

Assuming a program starts with the lines:
```
no1 = 2
v# = 41.09
```
what will be the result of the following instructions?

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

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

Unusual Calculations

Most of the results produced by these operators are easy to calculate manually as long as we are capable of basic arithmetic. However, when using AGK BASIC, the results of some statements are not quite so obvious. For example, we 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 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.
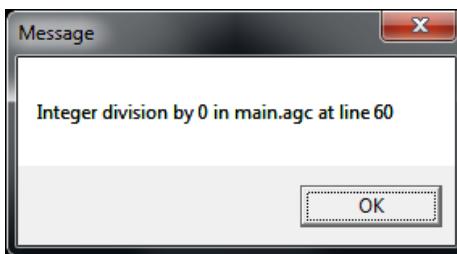
When using the division operator, a situation that we must guard against is division by zero. In mathematics, dividing any number by zero gives an undefined result, so most programming languages get quite upset if we try to get them to perform such a calculation. AGK BASIC will, when presented with a line such as

```
ans = 10/0
```

terminate the program and display a message such as that in FIG-4.10.

FIG-4.10

Division By Zero
Error Display

AGK BASIC reacts differently if a real number is involved in the calculation. For example, executing the line

```
Print(10.0/0)
```

displays the value

```
1.#INF00
```

We might be tempted to think that we would never write a division-by-zero statement,

but a more likely scenario is that our 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 problem.

Some statements may not appear to make sense if we 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.

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 multiplication and division with addition and subtraction performed last. The power operator (^) is said to have a **higher priority** than 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 the intermediate result:

```
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 4.8

What is the result of the calculation

```
12 - 5 * 12 / 10 - 5 ?
```

Using Parentheses

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

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

then 12+18 will be calculated first, leaving:

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

The next calculation is 30 / 9 :

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

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

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

Activity 4.9

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 are 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-4.11.

FIG-4.11

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
+	addition	4
-	subtraction	4

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 us 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       //*** subtract 1 from x ***
dec y, 3    //*** 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 bytecode than is possible when using the standard assignment approach.

FIG-4.12

The **inc** Statement

The format for the `inc` statement is shown in FIG-4.12.

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.

FIG-4.13

The **dec** Statement

The format for the `dec` statement is given in FIG-4.13.

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. If no number is given, then 1 is assumed.

Mod()

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

The `Mod()` function requires parameters, but it also supplies us with a result – the integer remainder produced by an integer division. When a function supplies a result, that value is known as a **return value**.

Syntax diagrams for functions that return a value have the format shown in FIG-4.14.

FIG-4.14

Functions that Return a Value

return type (Function Name) ( ) 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.

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

```
answer = Mod(7,3)
```

We could then use that value in a calculation or display it on the screen:

The `Mod()` function, returns the integer remainder produced after performing integer division on two values. The function's format is shown in FIG-4.15.

FIG-4.15

The **Mod()** Function

integer (Mod) ( ) v1 (,) v2 ( )

where:

v1	is an integer value giving the dividend (also called the numerator).
v2	is an integer value giving the divisor (or denominator).

For example,

```
ans = Mod(9, 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:

```
Mod(6, 3)        returns  0
Mod(7, 9)        returns  7
Mod(123, 10)     returns  3
```

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

```
Mod(-11, 3) returns -2
```

Variable Range

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

```
no1 = 123456789000
```

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

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

Integer values in AGK BASIC can be in the range -2,147,483,648 to +2,147,483,647 while floating-point values can be stored to about 7 significant figures.

✎ We can only say *about* since the actual value is stored in binary, not decimal.

String Operations

As well as being the arithmetic addition operator, the + symbol 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 4.12

What value will be stored as a result of the statement

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

The Print() Statement Again

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

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

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

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

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

```
Print("Hello " + "again") //***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*. Float and string variables can be displayed in the same way. Hence the lines

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

will produce the output

```
Charlotte
95.3
```


Activity 4.13

A program contains the following lines of code:

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

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

\n is known as an **escape character**. Escape characters are handled differently from standard characters.

If a string is placed in single quotes, the character combination \n can be used to force the cursor on to a new line. Hence, the line

```
Print('abc\ncde')
```

displays

```
abc
def
```

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

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

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

produces the following output on the screen:

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

> **Activity 4.14**
>
> Start a new project called *Name.*
>
> Have the program set the contents of the variable *name$* to *Jaqueline McKinnon* and then use output statements that display the contents of *name$* 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 4.15**
>
> 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 methods and we'll be looking at each of those methods later in the book.

Another way to retrieve information is to access the built-in hardware devices such as the timer.

AGK offers a few timer options. One gives us access to the time our program has been running to the nearest fraction of a second. Another gives the same information but to the nearest second. A third gives the time to the nearest one thousandth of a second.

Timer()

The `Timer()` function returns the time our program has been running in seconds and fractions of a second.

The syntax diagram for the `Timer()` statement is shown in FIG-4.16.

FIG-4.16

Timer()

float `Timer` ( )

The diagram tells use that the `Timer()` function returns a floating-point value and that no parameters are required by the function.

We could display a 'live' time by placing the statements

```
time# = Timer()
Print(time#)
```

in a program's `do...loop` structure.

Notice that the parentheses must be included when calling the function even though no information is placed within them.

Activity 4.16

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

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

Compile and run the program.

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

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

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

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

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

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

Activity 4.17

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

Test your modified program.

Activity 4.18

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

What difference does this make to what is displayed?

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

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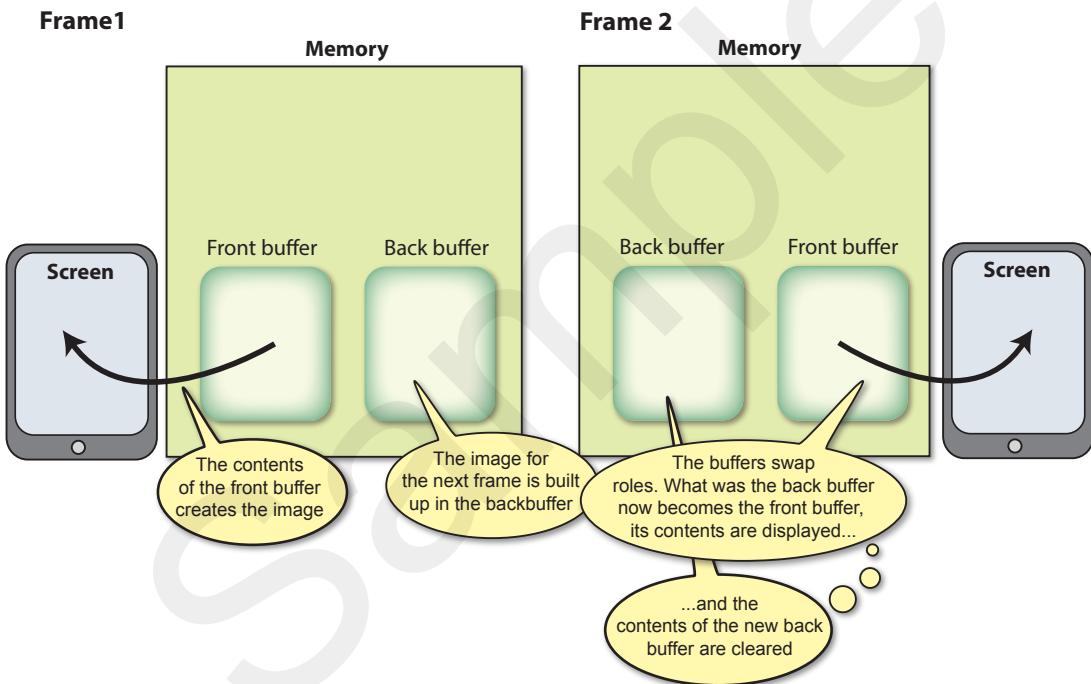
Let's take a moment out to get a deeper understanding of how `Sync()` works.

The contents of our screen are updated every time `Sync()` is executed. With `Sync()` inside the `do...loop` structure, this means the screen is likely to be updated many times per second. Each update redraws the entire contents of the screen. Each redrawing is known as a **frame**.

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

FIG-4.17 shows how these buffers are used in the construction of a frame.

FIG-4.17 How the Screen Display is Produced

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

When a `Sync()` statement is executed, the two areas of memory swap roles: what was the back buffer becomes the front buffer and its contents appear on the screen; and what was the front buffer becomes the back buffer and its contents are cleared.

It should be noted that handling the video buffers is not the `Sync()` statement's only purpose since it also updates various other aspects of an application. We will examine these other aspects of `Sync()` in later chapters.

Understanding the role of the buffers will give us some insight as to how the placement
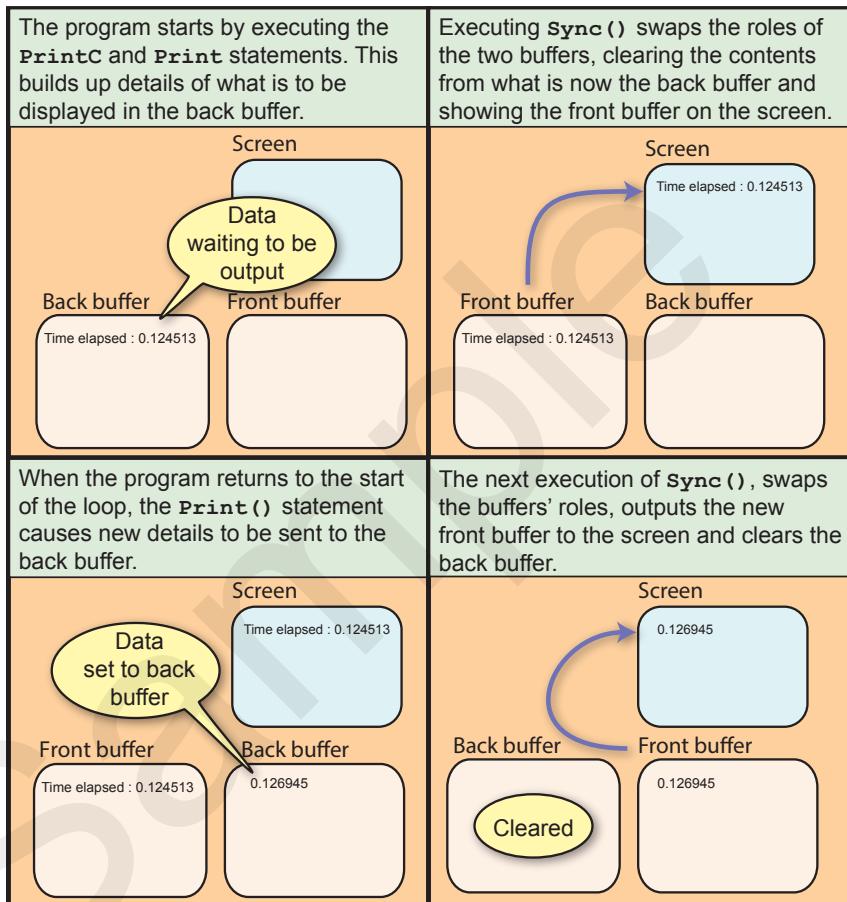
of the `Print()` and `PrintC()` statements affects the display produced by the *Time* project.

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 we won't have time to see it. After that, only the time appears.

FIG-4.18 explains the process involved when the first `PrintC()` statement appears before the `do`.

FIG-4.18

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 we want text to stay on the screen, we 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
```

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

The Official App Game Kit Tutorial Guide: Data

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

```
seconds = Mod(total_seconds, 60)
```

The final version of our program is shown in FIG-4.19.

```
// Project: Timer
// Created: 2015-01-03

// *** Set window properties ***
SetWindowTitle("Timer")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)


// *** Display time elapsed in mins and secs ***
do
    //*** Get time elapsed to nearest second ***
    total_seconds = Timer()

    //*** Convert to minutes and seconds ***
    minutes = total_seconds / 60
    seconds = Mod(total_seconds, 60)

    //*** Display the result ***
    PrintC("Time elapsed : ")
    PrintC(minutes)
    PrintC(":")
    Print(seconds)
    Sync()
loop
```

Activity 4.19

Modify your *Time* program to match the code given in FIG-4.19.

Compile and test your code.

ResetTimer()

Although the timer automatically starts tracking time from the moment our program begins executing, we can reset that timer to zero using the `ResetTimer()` function (see FIG-4.20).

Notice that this statement has neither 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, we can have that same value in milliseconds by using the `GetMilliSeconds()` function (see FIG-4.21).

GetSeconds()

If we are only interested in the time elapsed to the nearest second (without the fractional part), we can use the `GetSeconds()` function rather than `Timer()`. `GetSeconds()` has the format shown in FIG-4.22.

integer GetSeconds ( )

Activity 4.20

Modify *Time* to use `GetSeconds()` instead of `Timer()`.

Test your new code.

Some slight inaccuracy can creep into all the timing functions after a program has been running for some time, but if all we are interested in is time to the nearest second, there should never be any problem.

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 before another call to `Sync()` is made.

Halting a program for a specific time is achieved using the `Sleep()` function (see FIG-4.23).

Sleep (millisecs)

where:

millisecs   is an integer value giving the time in milliseconds for which the program execution is to halt.

Activity 4.21

Modify your *Time* program adding the line

```
    Sleep(2000)        // *** halt for 2 seconds ***
```

immediately after the line containing `Sync()`.

Run the program. How has the new line affected the program?

Another possible reason for using `Sleep()` – at least in a simple program – is to cause the output produced by a `Print()` or `PrintC()` statement, which is not in the `do...loop` structure to be displayed for sufficient time as to be visible to the user.

For example, the program in FIG-4.24 attempts to display the message

Program Starting

before going on to display the time the program has been running.

```
// Project: Message
// Created: 2015-01-05

//*** Set window properties ***
SetWindowTitle("Message")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)

//*** Display start up message ***
Print("Program Starting")
Sync()

do
    //*** Display time program running ***
    PrintC("Program has been running for ")
    PrintC(GetSeconds() )
    Print(" seconds")
    Sync()
loop
```

Activity 4.22

Create a project called *Message* containing the code shown in FIG-4.24.

Run the program. Does the initial message, *Program Starting* appear?

It shouldn't come as any surprise that the message is not visible since we have a situation similar to that we have already encountered in Activity 4.18.

However, one difference is that we have added a `Sync()` statement immediately after the first `Print()` statement. This forces the screen to be updated (as explained earlier) and therefore outputs the starting message. But the next `Sync()` statement (inside the `do...loop` structure) is executed too soon to allow us to see that first message. However, if we were to add a `Sleep()` statement immediately after the first `Sync()` statement, the program would halt long enough for us to view the message.

Activity 4.23

Modify *Message* by adding the line

```
Sleep(2000)
```

immediately after the first `Sync()` statement.

Run the program. Is the initial message, *Program Starting* now visible?

Generating Random Numbers

Often in a game we need to throw 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()` function is used to generate a random number between lower and upper limits (see FIG-4.24).

integer (Random) ( ) [low , high] ( )

where

low	is a non-negative integer giving the lowest value allowed.
high	is a non-negative integer giving the highest value allowed (maximum value allowed is 65,535).

The statement returns a random integer value in the range *low* to *high*. For example, if we wanted to simulate the throw of a die, we could write

```
dice_throw = Random(1,6)
```

which would store a random value between 1 and 6 in *dice_throw*.

Activity 4.24

Start a new project (*Dice*) and create code to perform the following logic:

    Throw a six-sided die
    Display the value thrown

Test the program by running it several times.

Save and close the project.

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-4.25 shows another use of the `Random()` statement to create a random background colour for the app window.

```
// Project: Background
// Created: 2015-01-03


// *** Set window properties ***
SetWindowTitle("Background")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)

//*** Cycle through random background colours ***

```

FIG-4.25
(continued)

Random Background
Colour

```
do
    //*** Generate a random value for each colour ***
    red = Random(0,255)
    green = Random(0,255)
    blue = Random(0,255)
    //*** Clear the screen using the new colour ***
    SetClearColor(red,green,blue)
    Sync()
loop
```

Activity 4.25

Start a new project (*Background*) and enter the code given in FIG-4.25.

What happens when you run the program?

Immediately after the `Sync()` statement, add the lines

```
//*** 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?

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))
```

Activity 4.26

Modify your *Background* project eliminating the need for the *red*, *green* and
blue variables. Test your program to ensure it still works correctly.

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 we don't know that algorithm, we 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 we want to use our own seed value, we can do so using the `SetRandomSeed()` statement. The most likely reason for doing this is to ensure we use the same seed value on each run and hence the same set of random values. Normally, of course, we 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-4.26.

where:

seed	is an integer value in the range {0.. 4,294,967,296 } which is used as the start-up for the formula used in the generation of pseudo random values.

Random2()

A second random number generator has been added to AGK BASIC. The need for a second number generator may not be obvious but the reason is to do with a weakness in most pseudo random number generators. If we call a generator often enough without reseeding it, it has a tendency to eventually begin repeating a sequence of numbers over and over again. For example, in the list of numbers below

5, 2, 1, 4, 4, 6, 5, 3, 6, 5, 3, 6, 5, 3, 6, 5, 3

we can see that after the first few numbers, the sequence 6, 5, 3 begins to repeat itself.

Of course, a random number generator won't start repeating until many hundreds or thousands of numbers have been generated and the sequence is likely to contain many more than the three values given above.

The `Random2()` function has the same basic format as `Random()` (see FIG-4.27), but creates many more numbers before running into any danger of creating a repeating sequence.

where

low	is an integer (-2,147,483,648 to 2,147,483,647) giving the lowest value allowed.
high	is an integer (-2,147,483,648 to 2,147,483,647)giving the highest value allowed.

When the parameters are omitted, numbers in the range -2,147,483,648 to 2,147,483,647 are generated. This is a much larger range than that produced by `Random()`.

SetRandomSeed2()

The `Random2()` statement has its own seeding function, `RandomSeed2()` (see FIG-4.28).

FIG-4.28

SetRandomSeed2()

$$\boxed{\text{SetRandomSeed2}} \; \boxed{(\,)} \; \boxed{\text{seed}} \; \boxed{)}$$

where:

  seed   is an integer value in the range {0.. 4,294,967,296 }
                    which is used as the start-up for the formula used in the
                    generation of pseudo random values.

Activity 4.27

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-4.29).

FIG-4.29

RandomSign()

$$\text{integer} \; \boxed{\text{RandomSign}} \; \boxed{(\,)} \; \boxed{\text{value}} \; \boxed{)}$$

where:

  value   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
                    value was 12 then the returned value will be either 12 or -12.
                    Each return option has a 50% chance of occurring.

User Input

For many games, the most important method of obtaining data is from the user. The game player, will be controlling a game by moving a mouse or joystick, or tapping on the screen. There is little need to type in information except perhaps a name when a high score is achieved.

AGK has statements available for handling all of these input methods 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 at a keyboard 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 downloads for this chapter. You'll find the relevant files in the folder *AGK2/Resources/ Ch04/* of the ZIP file you downloaded from the website. The file *Buttons.agc* contains two functions. These are:

  SetUpButtons()   This function sets up 12 circular buttons on the right of
                        the app window. The buttons are labelled 0 to 9,
                        ⇦*(Backspace)* and ⏎*(Enter)*.
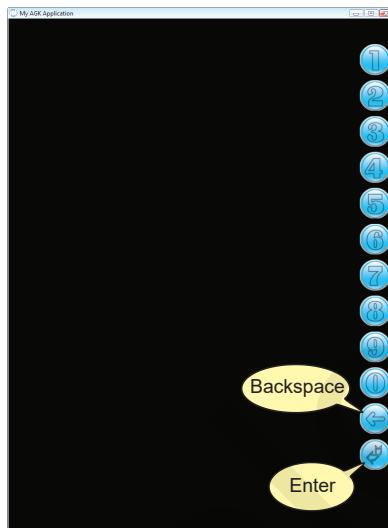
  GetButtonEntry() This function allows us to type in an integer value using

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-4.30.

FIG-4.30

Buttons Layout

The buttons are placed along the right edge to make them easy to press when the app is being used on a hand-held device. If we want to use these new functions in any of our projects, we have to follow a few simple steps. These are shown in FIG-4.31.

FIG-4.31

Using the Buttons

We start by creating a new project (*ButtonTest*) in which to test the button routines. Compiling the default code creates a *media* subfolder.	The ZIP file download for **Hands On AGK 2** contains a folder called *Chapter4*. This folder contains 3 files.
Compile to create *media* folder	Files in *Chapter4* folder
The PNG and TXT files are copied to the project's *media* folder. The AGC file is copied to the project's main folder.	In the project *main.agc* file, add an instruction to the compiler to include *Buttons.agc* as part of the project.
	File to be included **#include "Buttons.agc"**

The complete code (with comments) for *main.agc* is shown in FIG-4.32.

FIG-4.32

Button Input

```
// Project: TestButtons
// Created: 2015-01-03

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Test Buttons")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768.0/1024)

//*** Display the buttons ***
SetUpButtons()

//*** Get an integer value from the buttons ***
value_entered = GetButtonEntry()

do
    //*** Display the value entered ***
    PrintC("You entered ")
    Print(value_entered)
    Sync()
loop
```

Notice that the window size and display aspect have been changed to create a portrait-oriented window. This best suits the button layout.

#include

This is an instruction to the compiler to include the named source code file to a project. Typically, the added file will contain user-created functions (as is the case here) which have been created separately to be used in various future projects. This idea will be covered in detail in a later chapter.

Activity 4.28

Start a new project called *TestButtons*.

Compile the default project in order to create the *media* subfolder.

From the downloaded material in the *AGK2/Support/Ch04* folder, copy *Buttons.png* and *Buttons subtext.txt* into the *TestButtons* project's *media* folder.

From the *Chapter4* 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-4.32.

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.

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*
> Set the main window to be in portrait mode
> Call the functions as required by the program logic

Keeping Track of Variable Names
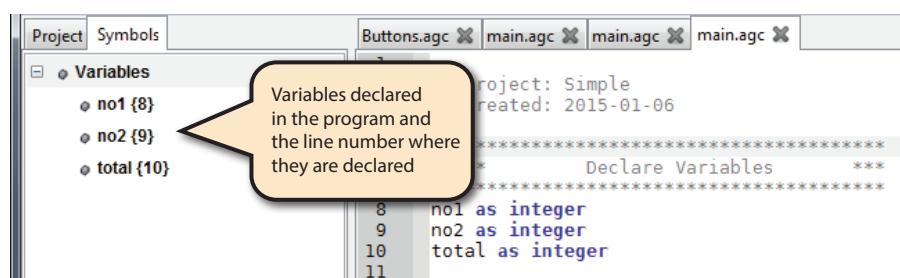
As our programs get more complex, we will almost certainly begin to use a considerable number of variables. In the past, this meant we needed a good memory in order to remember the name of a variable we last accessed a few hundred lines of code earlier.

Luckily, AGK BASIC's new IDE does a lot of the work for us. In the left-hand side of the window, where we've previously seen the list of projects currently open, we can click on the *Symbols* tab. Here is a list of all the variables we've explicitly declared within the program. Unfortunately, variables created on the fly (that is, without first being declared) are not listed – another reason to declare variables!

A simple example is shown in FIG-4.33.

FIG-4.33

Symbols Listing

Summary

- The assignment statement takes the form

        variable = value

    where *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:

        ^ * / + -

- Division involving two integer values always returns an integer result.

- Division involving at least one real value returns a real result.

- Division by zero is an error.

- 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 the next highest, followed by addition and subtraction.

- Terms enclosed in parentheses are always performed first.

- The **+** operator can be used to join strings.

- The **inc** operator adds a specified value to a variable.

- The **dec** operator subtracts a specified amount from a variable.

- The **Mod()** function returns the integer remainder produced after performing an integer division.

- Use **Timer()** to discover the time a program has been running. The result is in seconds and fractions of a second.

- Use **GetSeconds()** to discover the time a program has been running to the nearest whole second.

- Use **GetMilliSeconds()** to discover how long a program has been running in milliseconds.

- Use **Sleep()** to have a program stop for a given number of milliseconds.

- AGK uses a pseudo random number algorithm to create apparently random numbers within a specified range.

- The values generated are determined by an initial seed value.

- The default seed value for the algorithm is taken from the system's clock.

- Use **SetRandomSeed()** to set the seed value for the random number generator to a specified value.

- Use **Random()** to generate a random number. A range may be specified.

- If a great many random numbers are to be generated it is possible that the algorithm may cycle repeatedly through a set sequence of values.

- Use **Random2()** to reduce the chances of a repeated sequence of values or to increase the range of possible values.

- Use **SetRandomSeed2()** to specify a seed value for **Random2()**.

- Use `RandomSign()` to assign a random sign (- or +) to a specified numeric value.

- The `Sync()` function makes use of two screen buffers.

- The front buffer contains the data of the current screen output.

- The back buffer contains the data for the next screen output.

- Calling `Sync()` causes the two screen buffers to swap functions and clears the newly designated back buffer.

- Use `#include` to include another source code file in your project (the file must be in the current project's folder).

Testing Sequential Code

Every program we write needs to be tested. For a simple sequential program (such as those we have created so far) which accepts user input and produces an output, testing requires us to think of a value to be entered, predict what result this value should produce, and then run the program to check that we do indeed obtain the expected result from that test data.

The program below (see FIG-4.34) reads in a value from the buttons and displays the square root of that value.

```
// Project: SquareRoot
// Created: 2015-01-05
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("SquareRoot")
SetWindowSize(768, 1024, 0)
//*** Set display properties ***
SetDisplayAspect(768.0/1024)
SetOrientationAllowed(1, 1, 1, 1)

//*** Clear the screen ***
ClearScreen()

//*** Display buttons ***
SetUpButtons()

//*** Display prompt ***
Print("Enter a number : ")
Sync()
Sleep(2000)

//*** Get value ***
no = GetButtonEntry()

//*** Calculate square root ***
sqroot# = no^0.5

do
    rem *** Display result ***
    PrintC("Square root of ")
    PrintC(no)
    PrintC(" is ")
    Print(sqroot#)
    Sync()
loop
```

Activity 4.30

Start a new project called *SquareRoot* and compile the default code.

Copy *Buttons.agc* into the project's main folder. Copy *Buttons subimages. txt* and *Buttons.png* into the *media* folder. Recode *main.agc* to match the code given in FIG-4.34.

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 4.31**
>
> 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 we 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 we work out the expected result before entering our test data into the program – otherwise we have no way of knowing if the results we are seeing on the screen are correct.

> **Activity 4.32**
>
> What results would you expect from *SquareRoot* if your test data was 0 and -9?
>
> Attempt to run the program with these test values and check that the expected results are produced.

When a program requires a string value to be entered by the user, 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 we are testing.

Solutions

Activity 4.1

a) Integer	b) String	c) Integer	d) Float
e) String	f) Integer	g) Float	h) String
i) String	j) Float		

Activity 4.2

a)	-12	integer constant
b)	Elizabeth	string constant
c)	4.14	float constant
d)	27.0	float constant

Activity 4.3

a) Valid.
b) Invalid. Stores 13 since b is an integer variable.
c) Invalid. Not a string variable
d) Invalid. Remove $ from variable name or put quotes round the 5.
e) Valid. Single or double quotes are accepted.
f) Valid.

Activity 4.4

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 4.5

a) `desc$="tall"`
b) `result#= 12.34`

Activity 4.6

a) Valid.
b) Valid. but fraction part rounded and integer stored.
c) Invalid. A string cannot be copied to an integer variable.
d) Valid. The integer value in *no1* will be copied to *ans#* where it will be stored in floating-point format.
e) Invalid. A float cannot be copied to a string variable
f) Invalid. A string cannot be copied to a float variable

Activity 4.7

a) *no2* is 16
b) *x#* is 82.18
c) *no3* is zero (integer division)
d) *no4* is 9
e) *m#* is 0.0
f) *v2#* is 40.99
g) *no1* is 3
h) *no5* is -2

Activity 4.8

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 4.9

Steps:
8 * (6 - 2) / (3 - 1)
8 * 4 / (3 - 1)
8 * 4 / 2
32 / 2
16

Activity 4.10

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 4.11

a) 2
b) -1
c) 5
d) -4

Activity 4.12

term$ will hold the string *abc123xyz*

Activity 4.13

Output:
number
23

Activity 4.14

Code for *Name*:

```
name$ = "Jaqueline McKinnon"
do
   PrintC("Hello, ")
   PrintC(name$)
   Print(", how are you today?")
   Sync()
loop
```

Note the spaces inside the quotes to make sure there are gaps either side of the name.

Activity 4.15

Modified code for *Name*:

```
name$ = "Jaqueline McKinnon"
do
   Print("Hello, "+name$+", how are you today?")
   Sync()
loop
```

Activity 4.16

Modified code for *Time*:

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

The time displayed on the screen now updates continuously.

Activity 4.17

Modified code for *Time*:

```
do
    //*** Display time passed ***
    PrintC("Time elapsed : ")
    Print(Timer())
    Sync()
loop
```

Activity 4.18

Modified code for *Time*:

```
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 4.19

No solution required.

Activity 4.20

Modified code for *Time* (the modified section is highlighted):

```
// Project: Timer
// Created: 2015-01-03

//*** Set window properties ***
SetWindowTitle("Timer")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)


//*** Display time elapsed in mins and secs ***
do
    //*** Get time elapsed to nearest second ***
    total_seconds = GetSeconds()

    //*** Convert to minutes and seconds ***
    minutes = total_seconds / 60
    seconds = Mod(total_seconds, 60)

    //*** Display the result ***
    PrintC("Time elapsed : ")
    PrintC(minutes)
    PrintC(":")
    Print(seconds)
    Sync()
loop
```

Activity 4.21

Modified code for *Time*:

```
// Project: Timer
// Created: 2015-01-03

//*** Set window properties ***
SetWindowTitle("Timer")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)

//*** Display time elapsed in mins and secs ***
do
    //*** Halt for 2 seconds ***
    Sleep(2000)
    //*** Get time elapsed to nearest second ***
    total_seconds = GetSeconds()
```

```
    //*** Convert to minutes and seconds ***
    minutes = total_seconds / 60
    seconds = Mod(total_seconds, 60)

    //*** 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 4.22

The message is not visible.

Activity 4.23

Modified code for *Message*:

```
// Project: Message
// Created: 2015-01-05

//*** Set window properties ***
SetWindowTitle("Message")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)

//*** Display start up message ***
Print("Program Starting")
Sync()
Sleep(2000)

do
    //*** Display time program running ***
    PrintC("Program has been running for ")
    PrintC(GetSeconds() )
    Print(" seconds")
    Sync()
loop
```

The initial message now appears for two seconds before disappearing.

Activity 4.24

Code for *Dice*:

```
// Project: Dice
// Created: 2015-01-03

//*** Set window properties ***
SetWindowTitle("Dice")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)

//*** Throw dice ***
dice = Random(1,6)
do
    //*** Display value thrown ***
    PrintC("Value thrown was : ")
    Print(dice)
    Sync()
loop
```

Activity 4.25

The colours change so quickly that there may not be enough time to update the whole background before the colour changes again. This will cause bands of colour to appear.

Modified code for *Background*:

```
// Project: Background
// Created: 2015-01-03

// *** Set window properties ***
SetWindowTitle("Background")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)

//*** Cycle through random background colours ***
```

```
    do
        //*** Generate random value for each colour ***
        red = Random(0,255)
        green = Random(0,255)
        blue = Random(0,255)

        //*** Clear the screen using the new colour ***
        SetClearColor(red,green,blue)
        Sync()
        //*** 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 4.26

Modified code for *Background*:

```
// Project: Background
// Created: 2015-01-03

// *** Set window properties ***
SetWindowTitle("Background")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)

//*** Cycle through random background colours ***
do
    //*** Clear the screen using the new colour ***
    SetClearColor(Random(0,255),Random(0,255),
    ⮑Random(0,255))
    Sync()
    //*** Wait 0.5 seconds ***
    Sleep(500)
loop
```

Note The symbol ⮑ is used to indicate the continuation of a single line of code.

Activity 4.27

Modified code for *Dice*:

```
// Project: Dice
// Created: 2015-01-03

//*** Set window properties ***
SetWindowTitle("Dice")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024 / 768.0)

//*** Seed generator ***     TO BE REMOVED
SetRandomSeed(12)            //TO BE REMOVED

//*** Throw dice ***
dice = Random(1,6)

do
    //*** Display value thrown ***
    PrintC("Value thrown was : ")
    Print(dice)
    Sync()
loop
```

The program always generates a 6.

Activity 4.28

No solution required.

Activity 4.29

Reload your *Dice* project.
Modify the *startup.agc* file setting the width to 768 and the height to 1024.
Copy *Buttons.png* and *Buttons subtext.txt* into the project's *media* 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.

Modified code for :

```
// Project: Guess
// Created: 2015-01-03

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Guess")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Generate number (0 to 9) ***
dice = Random(0,9)

//*** Display user prompt ***
PrintC("Guess what my number is : ")
Sync()
Sleep(2000)

//*** Get an integer value from the buttons ***
guess = GetButtonEntry()

do
    //*** Display number generated  ***
    PrintC("My number was : ")
    Print(dice)
    PrintC("Your guess was : ")
    Print(guess)
    Sync()
loop
```

Activity 4.30

Start a new project called *SquareRoot*.
Compile the project to create the *media* folder.
Copy *Buttons.png* and *Buttons subtext.txt* into the project's *media* folder.
Copy *Buttons.agc* into the project's main folder.
Change the contents of *main.agc* to match that given in FIG-4.34.
Compile the program.

Activity 4.31

Running the program using the value of 16 gives the result 4.0.

Activity 4.32

The expected result using the value zero would be zero.

Using -9 would result in an error since negative values do not have a square root.

However, our on-screen buttons do not offer a minus sign, so we have (accidentally) created a user interface which makes it impossible to enter invalid values!

5

Selection

In this Chapter:

❏ **if...endif** Statement

❏ **Conditions**

❏ **Relational Operators**

❏ **Boolean Operators**

❏ **if...then** Statement

❏ **Nested if Statements**

❏ **select** Statement

❏ **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 = number THEN
    Say "Correct"
ENDIF
```

As we'll see, AGK BASIC also makes use of an `if` statement to handle such situations.

The `if` Statement

In its simplest form, the `if` statement in AGK BASIC takes the format shown in FIG-5.1.
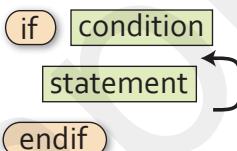
where:

> **condition**   is any term which can be reduced to a true or false value.
>
> **statement**   is any executable AGK BASIC statement.

The arrowed line within 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 statement 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-5.2.

where:

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	<>

value1 and **value2**   may be constants, variables, or expressions.

relational operator   is one of the symbols given in FIG-5.3.

FIG-5.3

The Relational Operators

From our *condition* 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 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 5.1

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`
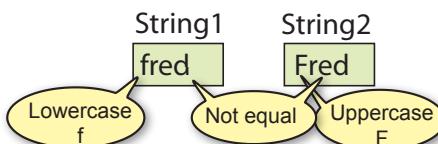
When two strings are checked for equality as in the condition

```
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-5.4).

FIG-5.4

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, we will occasionally represent spaces within a string using a triangle symbol. This means that rather than show the contents of a string as

Hello world

you may see

HelloΔworld

This is only done when clarification of the exact contents of a string is required. For example, the strings *hello* and *hello*Δ 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 equal, as in the conditions

```
if name$ = "Fred"
if village$ <> "Drummore"
```

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 binary code used within the computer to store a *B* has a greater numeric value than the code used to store *A*. And, although the coding used means that the order of the letters' values match their order in the alphabet, there are differences. For example, all lowercase letters have a higher numeric value than any uppercase letter. Hence, *z* is greater than *Z*, *M*, or *A*.

The method of coding characters is known as UTF-8 and is equivalent to the older ASCII (American Standard Code for Information Interchange) system for the original character set. 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 will be less than another if that first string appears first in an alphabetically ordered list. Hence,

"Aardvark"   is less than   "Abolish"

But remember to watch out for upper and lower case differences as in

```
"Aardvark" < "aardvark"
```

which is true since *A* is less than *a*.

If two strings differ in length, with the shorter matching the first part of the longer as in

```
"abc" < "abcd"
```

then the shorter string is considered to be *less than* the longer string.

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 UTF-8 and ASCII coding systems.

Activity 5.2

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 to programming code. 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 the required result. 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 Mod(number, 2) = 0
    Print("Even number")
endif
```

> **Activity 5.3**
>
> Start a new project *EnglishToCode*. The program will accept values from the screen buttons we used in previous programs. 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.

Since we are always interested in creating efficient algorithms, the slight problem with the solution to Activity 5.3 is that the `if` statement is inside the `do...loop` structure. And, although the `Print()` statement must be there to have the message remain on the screen, it seems inefficient to have the `if` statement there too, since we know the condition, after being tested once, will always return the same result.

To get round this, we can change the logic of the program slightly as follows:

```
Read in values for no1 and no2
Set message to an empty string
IF no1 is exactly divisible by no2 THEN
    Set message to "Exactly divisible"
ENDIF
Print message
```

Now, only the last line of the algorithm needs to be within the `do...loop` structure.

Using `if`

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
```

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!")
endif
```

The statement `Print("You win!")` 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-5.5.

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!")
endif
```

All possible combinations for two conditions linked by an `or` are shown in FIG-5.6.

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 us changing the `if` statement given above to

```
if total = 7 or 11
```

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 5.7

Start a new project called *TwoDice*. Create a program using the two-dice code given above to display *You win!* when the dice total is 7 or 11.

Add statements to display the values thrown on the two dice. This should appear irrespective of the values thrown.

Test your program.

A compound 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` – `not` is covered in the next section) 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 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
```

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 5.9, 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.

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-5.7.

FIG-5.7

Boolean Priority

Operator	Priority
( )	1
not	2
and	3
or	4

else - Creating Two Alternative Actions
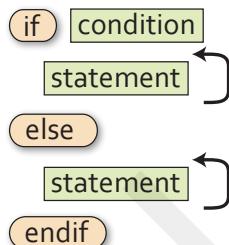
Like structured English, AGK BASIC offers an **else** extension to the basic **if** statement where we can specify any actions to be taken only when the specified condition is false. For example, we can add the word **else** to our original **if** statement in the guessing game to allow for two alternative messages:

```
if guess = number
    Print("Correct")
else
    Print("Wrong")
endif
```

This gives us the longer version of the **if** statement format as shown in FIG-5.8.

FIG-5.8

if...else...
endif

Note that we can have an unlimited number of statements between **else** and **endif**.

Activity 5.12

In your *Guess* 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 *number* and *guess* values.

Test and save your program.

Activity 5.13

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 logic below:
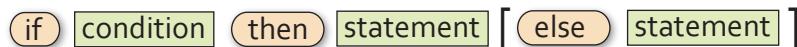
```
Display a prompt message for first number
Read the first number
Display a prompt message for the second number
Read the second number
IF first number is less than the second number THEN
        Set answer to first number
ELSE
        Set answer to second number
ENDIF
IF answer is an even number THEN
        Set message to "Even"
ELSE
        Set message to "Odd"
ENDIF
Display answer and message
```

The Other `if` Statement

AGK BASIC actually offers a second version of the `if` statement which has the format shown in FIG-5.9.

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 5.12 would have to be written as:

```
if number = guess then Print("Correct") else Print("Wrong")
```

This lack of indented layout is enough to have the hardened programmer throw up his 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 5.14**
>
> 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 inverts 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 select one option from three or more alternatives? How can we create code to deal with such a situation?

In structured English we used 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 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 *Guess* 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 only two alternative messages, *Correct* or *Wrong,* and was coded as:

```
if guess = number
    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 > number
    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 = number
    Print("Correct")
else
    if guess > number
       Print("Your guess is too high")
    else
       Print("Your guess is too low")
    endif
endif
```

We have now created a nested `if` situation, where the `if guess > number` statement is inside the `else` section of the `if guess = number` statement.

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:

Your guess is too high	(guess is more than 2 above the number)
Your guess is slightly too high	(guess is no more than 2 above the number)
Correct	(guess equals the number)
Your guess is slightly too low	(guess is no more than 2 below the number)
Your guess is too low	(guess is more than 2 below the number)

✎

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 *Guess* 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
```

In a situation that included even more options, the indentation can be so extreme that you may reach the right-hand side of the screen! 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
```

As we can see, each option is given the same indention as the last, and the closing set of `endif` keywords placed on a single line. This gives a much neater layout which is still easy to follow.

> **Activity 5.18**
>
> Modify the layout of your *Guess* program to conform to this new layout style for multi-way selection. Retest 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 5.19**
>
> Modify *Guess* to use the `elseif` term. Retest 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 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.

```
//*** Generate number in the range 0 to 8 ***
day = Random(0,8)

//*** Display name of the day generated ***
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

//*** Display the value generated ***
Print(day)
```

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. After the code in the selected `case` option has been carried out, control moves to the instructions following the `endselect` statement.

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
```
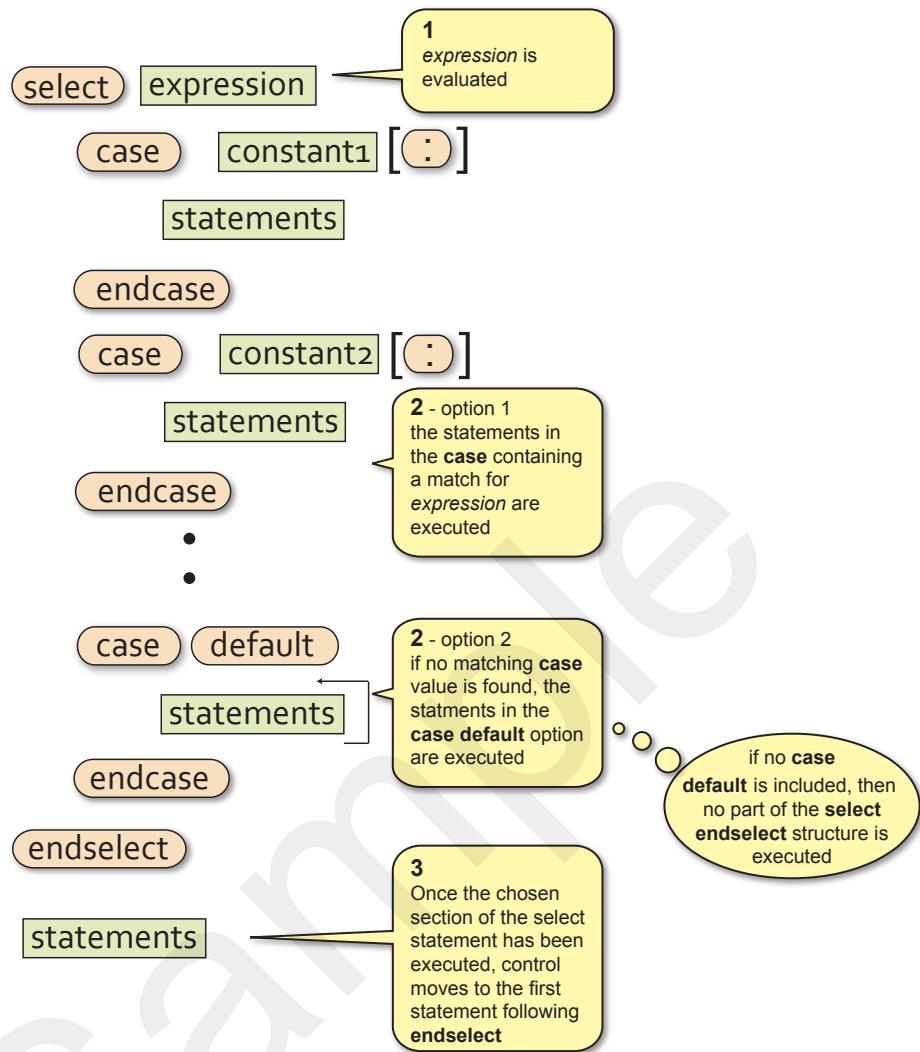
then, if a value outside the range 1 to 7 is generated, the statement in the `case default` option will be executed.

FIG-5.10 shows how the `select` statement is executed.

FIG-5.10

How select Works

select expression

> **1**
> *expression* is evaluated

case constant1 [:]

statements

endcase

case constant2 [:]

statements

> **2** - option 1
> the statements in the **case** containing a match for *expression* are executed

endcase

•
•

case default

statements

> **2** - option 2
> if no matching **case** value is found, the statments in the **case default** option are executed

> if no **case default** is included, then no part of the **select endselect** structure is executed

endcase

endselect

statements

> **3**
> Once the chosen section of the select statement has been executed, control moves to the first statement following **endselect**

Several values can be specified for each `case` option. If the value of the term given in the `select` statement matches any of the values listed in a `case` statement, then the statement(s) in that `case` option will be executed. For example, using the lines

```
num = Random(1,10)
select num
   case 1,3,5,7,9
      Print("Odd")
   endcase
   case 2,4,6,8,10
      Print("Even")
   endcase
endselect
print(num)
```

the word *Odd* would be displayed if any odd number between 1 and 9 was generated.

The values given beside the `case` keyword may also be strings as in the example below:

```
//*** Read a name ***
name$ = GetName()

//*** Respond to name entered ***
select name$
   case "Jack","Jill" :
      Print("Hello friend")
   endcase
   case default
      Print("I do not know your name")
   endcase
endselect
```

Although the **case** value may also be a real value as in the line
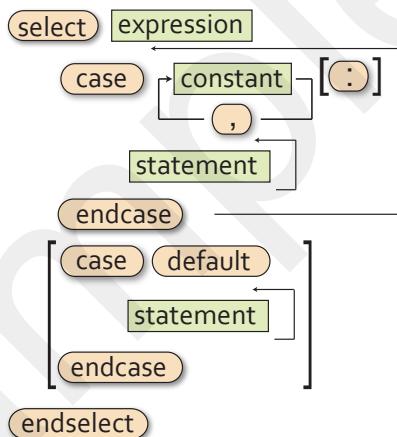
```
case 1.52
```

it is a bad idea to use real values since the machine cannot always store these accurately. If a float 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-5.11.

FIG-5.11

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 5.20

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 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, it would be impractical to list all the possible values for each group in a `case` line. Instead, we would have to code such a problem using nested `if` statements.

Testing Selection 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 results in the condition being false. Therefore, if a program contained the lines

```
no = GetButtonEntry()
if Mod(no, 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.

✎
This also applies to *less than or equal to* and *greater than or equal to* operators.

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 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(0,24) - 12
```

to

```
no = -7
```

Now we can run the program knowing which value is being used 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-5.10.

FIG-5.10

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
```

the theoritical combinations of true and false for the three conditions are as shown in FIG-5.11.

FIG-5.11

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 several of these combinations are impossible to achieve. The variable *total* cannot contain the values 7 and 11 at the same time (the conditions are mutually exclusive), so the last two combinations shown in the table cannot be achieved. Also *total* cannot have a value of 7 and *dice1* be equal to *dice2* (two identical values must sum to an even number). For the same reason *total* cannot be 11 and *dice1* = *dice2*. This eliminates two more combinations from the table.

So our test data will use test values which create only the remaining 4 combinations.

Activity 5.22

Suggest a set of test values for the latest version of the *Guess* project (Activity 5.19).

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 a nested `if` structure 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 5.1

    a) Valid.
    b) Valid.
    c) Valid.
    d) Invalid. => is not a relational operator (should be >=).
    e) Invalid. Integer variable compared with string.
    f) Invalid. *14 High Street* should be in quotes.

Activity 5.2

    a) False. Only the second string contains a space.
    b) True. "def"is shorter and matches the first three characters
        of "defg".
    c) True. "A" comes before "B".
    d) False. Only the second string contains a full stop.
    e) False. Only the second string contains a capital D.
    f) True. "*" has a greater ASCII coding than "&"

Activity 5.3

Code for *EnglishToCode*:

```
// Project: EnglishToCode
// Created: 2015-01-06

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("English To Code")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Get two values from the buttons ***
Print("Enter first number : ")
Sync()
Sleep(1000)
no1 = GetButtonEntry()
Print("Enter second number : ")
Sync()
Sleep(1000)
no2 = GetButtonEntry()

do
    //*** If no1 exactly divisible by no2, display
    ⤷message ***
    if Mod(no1,no2) = 0
      Print("Exactly divisible")
    endif
    Sync()
loop
```

Activity 5.4

Modified code for *EnglishToCode*:

```
// Project: EnglishToCode
// Created: 2015-01-06

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("English To Code")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Get two values from the buttons ***
Print("Enter first number : ")
Sync()
```

```
Sleep(1000)
no1 = GetButtonEntry()
Print("Enter second number : ")
Sync()
Sleep(1000)
no2 = GetButtonEntry()

//*** Set appropriate message ***
message$ = ""
if Mod(no1,no2) = 0
    message$ = "Exactly divisible"
endif

do
    //***Display message ***
    Print(message$)
    Sync()
loop
```

Activity 5.5

Modified code for *Guess*:

```
// Project: Guess
// Created: 2015-01-03

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Guess")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Generate number (0 to 9) ***
number = Random(0,9)

//*** Display user prompt ***
PrintC("Guess what my number is : ")
Sync()
Sleep(2000)

//*** Get an integer value from the buttons ***
guess = GetButtonEntry()

do
    //*** If guess isn't correct, display message ***
    if guess <> number
        Print("Wrong")
    endif
    //*** Display number generated  ***
    PrintC("My number was : ")
    Print(number)
    PrintC("Your guess was : ")
    Print(guess)
    Sync()
loop
```

Activity 5.6

The `if` structure in *Guess* becomes:

```
if guess <> number
    diff = number - guess
    PrintC("Wrong. You were out by ")
    Print(diff)
endif
```

You may get a negative value displayed when the guess is greater than the random number generated.

Activity 5.7

Code for *TwoDice*:

```
// Project: TwoDice
// Created: 2015-01-09

//*** Set window properties ***
SetWindowTitle("Two Dice")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024/768.0)
ClearScreen()
```

```
//*** Generate dice values ***
dice1 = Random(1,6)
dice2 = Random(1,6)
total = dice1 + dice2

do
    //*** If 7 or 11, display win message ***
    if total = 7 or total = 11
      Print("You win!")
    endif

    //*** Display value on dice ***
    PrintC("Dice 1: ")
    Print(dice1)
    PrintC("Dice 2: ")
    Print(dice2)
    Sync()
loop
```

Activity 5.8

The `if` statement in *TwoDice* should now read:

```
if total = 7 or total = 11 or dice1 = dice2
  Print("You win!")
endif
```

Activity 5.9

Code for *ThreeDice*:

```
// Project: ThreeDice
// Created: 2015-01-11

//*** Set window properties ***
SetWindowTitle("Three Dice")
SetWindowSize(1024, 768, 0)
SetDisplayAspect(1024/768.0)
ClearScreen()

// *** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)
dice3 = Random(1,6)

// *** If any two dice match set up message ***
mess$ = ""
if dice1 = dice2 or dice1 = dice3 or dice2 = dice3
    mess$ = "You win!"
endif

do
    //*** Display message ***
    Print(mess$)
    // *** Display values ***
    PrintC("dice 1: ")
    Print(dice1)
    PrintC("dice 2: ")
    Print(dice2)
    PrintC("dice 3: ")
    Print(dice3)
    Sync()
loop
```

Activity 5.10

a)   `if no1 >= 1 and no1 <= 12`

b)   `if no2 < 1 or no2 > 20`

c)   `if no1 <> 0 and no2/no1 > no3`

d)   `if no1 < 0 and (no2 > 0 or no3 > 0)`

e)   `if (Mod(no1,2) = 0 and Mod(no2,2) = 0) or`
     ↳`(Mod(no1,2) = 0 and Mod(no3,2) = 0) or`
     ↳`(Mod(no2,2) = 0 and Mod(no3,2) = 0)`

Activity 5.11

```
dice1 <> dice2 and dice1 <> dice3 and dice2 <> dice3
```

Activity 5.12

Modified code for *Guess* is:

```
// Project: Guess
// Created: 2015-01-011

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Guess")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Generate number (0 to 9) ***
number = Random(0,9)

//*** Display user prompt ***
PrintC("Guess what my number is : ")
Sync()
Sleep(2000)

//*** Get an integer value from the buttons ***
guess = GetButtonEntry()

do
    //*** Display response to guess ***
    if guess <> number
      Print("Wrong")
    else
      Print("Correct")
    endif
    //*** Display number generated  ***
    PrintC("My number was : ")
    Print(number)
    PrintC("Your guess was : ")
    Print(guess)
    Sync()
loop
```

Activity 5.13

Code for *TwoNumbers*

```
// Project: TwoNumbers
// Created: 2015-01-11

// *** Include Buttons functions ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Two Numbers")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

// *** Display buttons ***
SetUpButtons()

// *** Get numbers ***
Print("Enter first number ")
Sync()
Sleep(2000)
no1 = GetButtonEntry()
Print("Enter second number ")
Sync()
Sleep(2000)
no2 = GetButtonEntry()

// *** Determine smaller value ***
if no1 < no2
    answer = no1
else
    answer = no2
endif

// *** Determine if answer is odd or even ***
if Mod(answer,2) = 0
    mess$ = "This is an even number"
else
    mess$ = "This is an odd number"
endif

do
    //*** Display smaller ***
    PrintC("Smaller value is ")
    Print(answer)
```

```
    //*** Odd or even message ***
    Print(mess$)
    Sync()
loop
```

Activity 5.14

a) A Boolean expression is an expression whose result is either true or false.

b) Six. $<, <=, >, >=, =, <>$

c) **not** is performed first, **and** next and **or** last. This order changes if parentheses are used.

Activity 5.15

Modified code for *Guess*:

```
// Project: Guess
// Created: 2015-01-011

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Guess")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Generate number (0 to 9) ***
number = Random(0,9)

//*** Display user prompt ***
PrintC("Guess what my number is : ")
Sync()
Sleep(2000)

//*** Get an integer value from the buttons ***
guess = GetButtonEntry()

do
    //*** Respond to guess ***
    if guess = number
      Print("Correct")
      else
      if guess > number
        Print("Too high")
      else
        Print("Too low")
      endif
    endif

    //*** Display number generated  ***
    PrintC("My number was : ")
    Print(number)
    PrintC("Your guess was : ")
    Print(guess)
    Sync()
loop
```

Activity 5.16

Code for *RandomNumber*:

```
// Project: RandomNumber
// Created: 2015-01-11

//*** Set window properties ***
SetWindowTitle("Random Number")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()
// *** Generate number ****
no = RandomSign(Random(0,12))

// *** Set up message ***
if no < 0
    mess$ = "Negative"
else
    if no = 0
        mess$ = "Zero"
    else
        mess$ = "Positive"
    endif
endif
```

```
do
    //*** Display message ***
    Print(mess$)
    // *** Display number ***
    PrintC("Number : ")
    Print(no)
    Sync()
loop
```

Notice the use of `RandomSign()` to generate negative as well as positive values.

Activity 5.17

Modified code for *Guess*:

```
// Project: Guess
// Created: 2015-01-011

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Guess")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Generate number (0 to 9) ***
number = Random(0,9)

//*** Display user prompt ***
PrintC("Guess what my number is : ")
Sync()
Sleep(2000)

//*** Get an integer value from the buttons ***
guess = GetButtonEntry()

//*** Calculate difference ***
diff = number - guess

do
    //*** Respond to 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

    //*** Display number generated  ***
    PrintC("My number was : ")
    Print(number)
    PrintC("Your guess was : ")
    Print(guess)
    Sync()
loop
```

Activity 5.18

The multi-way selection section of *Guess*'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 5.19

New new multi-way selection coding in *Guess* 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 5.20

Code for *Days*:

```
// Project: Days
// Created: 2015-01-11

//*** Set window properties ***
SetWindowTitle("Days")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

// *** Generate value ***
day = Random(0,8)

do
    // *** 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
    // *** Display number generated ***
    Print(day)
    Sync()
loop
```

Activity 5.21

Code for *Cards*:

```
// Project: Cards
// Created: 2015-01-11

//*** Set window properties ***
SetWindowTitle("Cards")
SetWindowSize(768,1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

// *** Generate card value ***
card = Random(1,13)
```

```
do
    // *** Display card type ***
    select card
        case 11,12,13
            Print("Court card")
        endcase
        case default
            Print("Spot card")
        endcase
    endselect
    Print(card)
    Sync()
loop
```

Note that all spot cards can be handled in the `case default` option because there is no chance of an invalid value being used.

Activity 5.22

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 *number* and *guess* to be displayed.

Since the number values are randomly generated it would be impractical to use our test data. We can overcome this problem by setting the variable number to a specific value rather than determining its value using `Random()`. Once testing is complete, the random assignment can be restored.

⑥

Iteration and Debugging

In this Chapter:

❑ **while...endwhile** Structure

❑ **repeat...until** Structure

❑ **for...next** Structure

❑ **do...loop** Structure

❑ **Validating Input**

❑ **The exit Statement**

❑ **Testing Iterative Structures**

❑ **Using the Debugger**

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. The `while...endwhile` and `repeat...until` structures take a similar form to their structured English equivalent. The `for...next` structure performs the same purpose as structured English's FOR...ENDFOR but has a more complex syntax. The final construct, `do...loop`, has no equivalent in structured English.

The `while...endwhile` Construct

The `while` statement loop structure is identical in operation to the WHILE loop in structured English but drops the term DO.

This structure allows us to continually execute a section of code as long as a specified condition is being met. For example, back in Chapter 1 we described the rules for the dealer in the card game Blackjack as:

```
Calculate the value of the initial two cards in hand
WHILE value of cards in hand is less than 17 DO
    Take another card
ENDWHILE
```

This can be coded in AGK BASIC as:
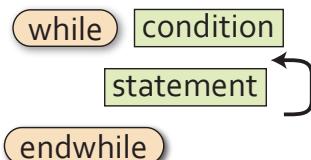
```
value = card1 + card2
while value < 17
    value = value + Random(1,10)
endwhile
```

Here the `Random(1,10)` term is used to simulate (not entirely accurately) the value of a new card.

The syntax of AGK BASIC's `while...endwhile` construct is shown in FIG-6.1.

`while....`
`endwhile`

where:

condition	is a Boolean expression and may include `and`, `or`, `not` and parentheses as required.
statement	is any valid AGK BASIC statement.
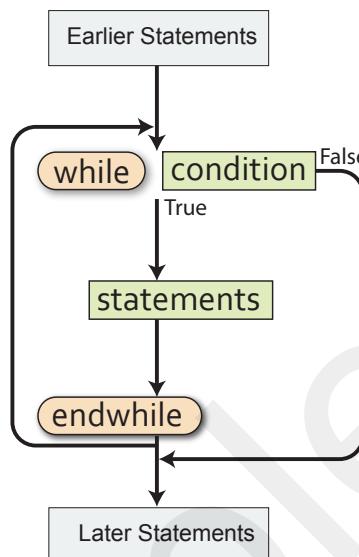
The `while...endwhile` construct is an **entrance-controlled** loop. That is, the condition at the start of the loop is tested and only if that condition is true, are the statements within the loop executed. When the `endwhile` term is reached, control

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-6.2.

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:

```
//*** Display user prompt ***
Print("Guess what my number was (0 to 9) : ")
Sync()
Sleep(2000)

//*** Get a guess in range 0 to 9 ***
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 6.1

Modify your *Guess* 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.

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
```

✎
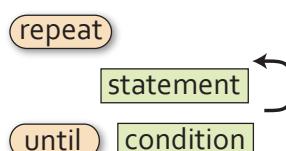
The code assumes we are using the *Button* routines introduced in the previous chapter to accept input.

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-6.3.

FIG-6.3

repeat...until

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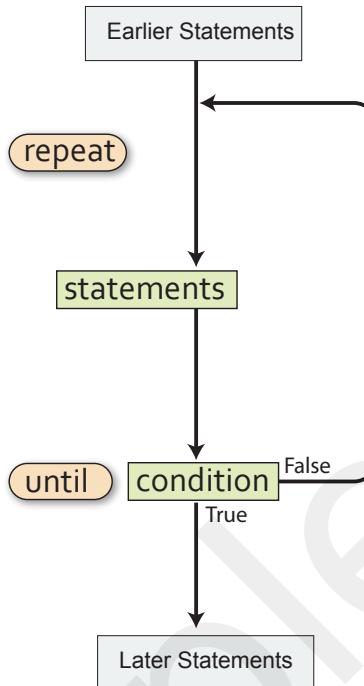
condition   is a Boolean expression and may include **and**, **or**, **not** and parentheses as required.

statement   is any valid AGK BASIC statement.

The operation of the `repeat...until` construct is shown graphically in FIG-6.4.

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 of these examples, the action specified between the FOR and ENDFOR terms will be executed a known number of times.

In AGK BASIC the `for...next` 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 automatically be assigned the value 1 when the `for` loop is about to start. Each time the statements within the loop have been executed, 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 6.5

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.

The code below makes use of a `for` loop to display 10 asterisks:

```
for k = 1 to 10
    print("*")
next k
Sync()
```

Activity 6.6

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 during the final iteration. 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
```

Here the loop counter, *c*, will start at 2 and then increment to 4 on the next iteration. The program in FIG-6.5 uses the **step** option to display the 7 times table from 1 x 7 to 12 x 7.

FIG-6.5

7 Times Table

```
// Project: Tables
// Created: 2015-01-12

//*** Set window properties ***
SetWindowTitle("7 Times Table")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display 7 times table ***
do
    for c = 7 to 84 step 7
        Print(c)
    next c
    Sync()
loop
```

Activity 6.7

Start a new project, *Tables*, that implements the code shown in FIG-6.5.

Test the program.

Modify the program so that it displays the 12 times table from 1 x 12 to 12 x 12.

Test your project.

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.

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. Looping will always stop before the variable passes the specified finish value.

The start, finish and step values of a `for` loop can be defined using a variable or arithmetic expression as well as a constant. For example, in FIG-6.6 below the user is allowed to enter the upper limit of the `for` loop.

FIG-6.6

Using a Variable in a **for...next** Statement

```
// Project: UserLoop
// Created: 2015-01-21


//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window title and size ***
SetWindowTitle("User Loop")
SetWindowSize(720,1024,0)
SetDisplayAspect(720.0/1024)

//*** Clear the screen ***
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Get upper limit ***
Print("Enter for loop upper limit : ")
Sync()
Sleep(2000)
high = GetButtonEntry()

//*** Display values 1 to high ***
do
   for c = 1 to high
      print(c)
   next c
   Sync()
loop
```

The program will display every integer value between 1 and the number entered by the user.

If this involves more than 30 numbers being displayed, there will not be space within the app window to show them all. Numbers greater than 30 are written to positions not visible within the program window. The contents of the window does not scroll and there are no scrollbars to allow access to a larger area.

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()
```

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 1.900000.

If the output were to display the values produced to more decimal places we'd
understand what was causing the anomaly. Although the final value displayed is
1.900000, a more accurate display would show c to contain the value 1.90000021458.

This difference is caused by rounding errors created when the compiler converts
from the decimal values that we use in the code to the binary values favoured by the
computer.

So, if c has a value of 1.90000021458, attempting to add 0.1 when the `for` loop
attempts another iteration, would take us past the 2.0 upper limit of the loop and
hence iteration stops.

The underlying cause of the problem is the fact that 0.1 (our step size) cannot be
represented accurately in binary. In fact, the binary value it uses is approximately
equivalent to the decimal value 0.10000002384.
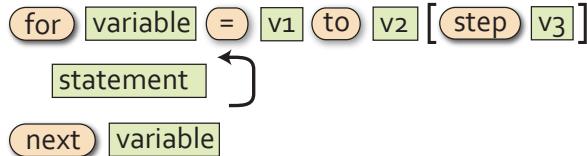
This time the program produces the output we might expect. And this is because the
value 0.25_{10} can be represented exactly in binary as 0.11_2.

The format of the `for...next` statement is shown in FIG-6.7.

FIG-6.7

`for...next`

where:

variable	is either an integer or float 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**.
v1	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.
v2	is the final value of the loop variable. The loop variable will usually contain this value the last time the loop body is executed.
v3	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.
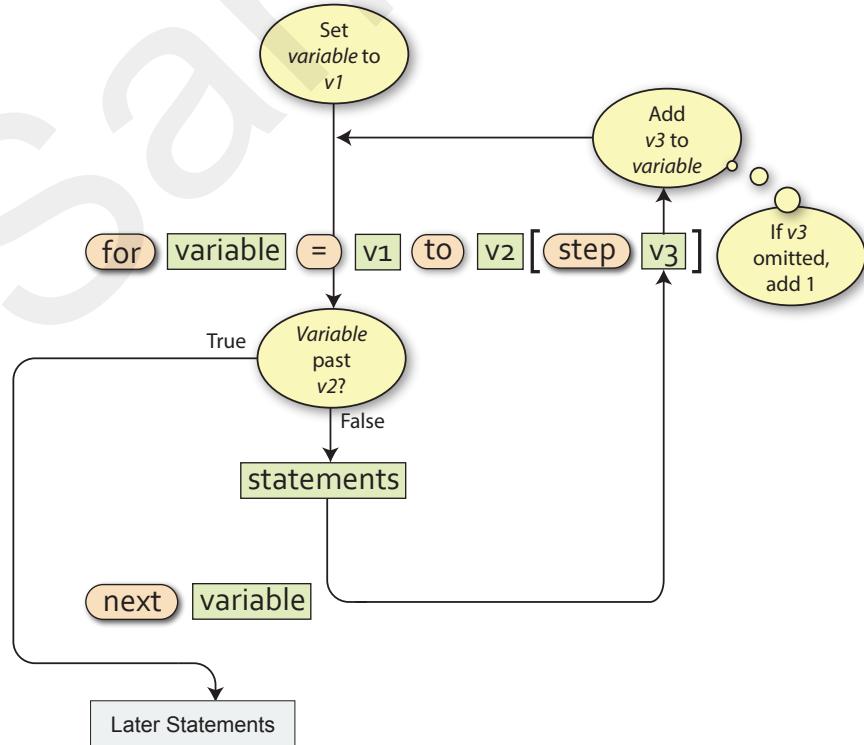
The operation of the `for...next` statement is shown graphically in FIG-6.8.

FIG-6.8

How **for...next**
Operates

Finding the Smallest Value in a List of Values

There are several tasks that will crop up over and over again in our 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 we managed to come up with the correct answer, we might struggle to give a verbal description of the strategy we used.

Now, let's imagine we wanted to record the coldest temperature achieved in our area during the current year. Since this involves a longer list of data which also takes a full year to access, we would have to come up with an organised way of getting the information we want. Perhaps we could 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, we can erase the previous record and write down this new temperature. By the end of the year our 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 number
Set smallest to first number
FOR 6 times DO
    Get number
    IF number < smallest THEN
        Set smallest to number
    ENDIF
ENDFOR
Display smallest
```

The **exit** Statement

FIG-6.9

exit

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-6.9.

$$\boxed{\text{exit}}$$

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 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-6.10.

FIG-6.10

Using **exit**

```
// Project: SumDice
// Created: 2015-01-21


//*** Set window properties ***
SetWindowTitle("Sum Dice")
SetWindowSize(1024,768,0)
SetDisplayAspect(1024.0/768)
ClearScreen()

//*** Set total to zero ***
total = 0
//*** for 5 times do ***
for c = 1 to 5
   //*** Display number of rolls so far ***
   PrintC("Roll number ")
   Print(c)
   Sync()
   Sleep(1500)
   //*** Throw both dice ***
   dice1 = Random(1,6)
   dice2 = Random(1,6)
   //*** Display throw number and dice values ***
   PrintC("dice 1 : ")
   PrintC(dice1)
   PrintC("        dice 2 : ")
   Print(dice2)
   Sync()
   Sleep(4000)
```

FIG-6.10
(continued)

Using **exit**

```
   //*** if either dice is a 1 then quit loop ***
   if dice1 = 1 or dice2 = 1
      exit
   endif
   //*** Add dice throws to total ***
   total = total + dice1 + dice2
next c

do
   //*** Display final score ***
   PrintC("Your final score was : ")
   Print(total)
   Sync()
loop
```

Activity 6.15

Create a new project call *SumDice* containing the code given in FIG-6.10.

Run the program and check that the loop exits if a 1 is thrown.

Modify the program so that the number of throws made is also displayed.

How many throws are reported if none of the throws result in a die showing 1?

The problem highlighted in Activity 6.15 arises because of the way in which the **for** loop operates (as shown in FIG-6.8). A **for...next** loop normally exits only after the loop counter has past the specified upper limit. So when we want the loop counter to range in value between 1 and 5, it actually takes on the value 6 before the loop is exited.

To solve this problem in our *SumDice* project we will need to add the lines

```
   if c = 6
      dec c
   endif
```

Activity 6.16

Modify *SumDice* so that it reports the correct number of throws when a 1 does not appear on any die.

Test your project.

Activity 6.17

Modify *Guess*, so that the program generates a number between 1 and 100 and the player is allowed up to seven guess to come up with the correct answer, but exits before all iterations are complete if a correct guess is achieved. The number of guesses required should also be displayed.

Test your project.

The `continue` Statement

The `continue` statement, like `exit`, is designed solely for use within a loop structure. Its effects are less severe than that of `exit`; rather than exit the loop structure entirely it exits the current iteration only, jumping back to the start of the loop where the next iteration of the loop proceeds as normal. The statement has the format shown in FIG-6.11.

(continue)

The program in FIG-6.12 demonstrates the effect of the `continue` statement. The program generates two random numbers in the range 0 to 20, performs integer division, dividing the first number by the second and then displays the random numbers and the result of the calculation. The `continue` statement is used to skip the calculation and display statements if the second value generated is zero (since division by zero is not allowed). Looping stops if the result of the calculation is 1.

Additional `Print` statements have been added to highlight what is happening when the program runs.

FIG-6.12

Using **continue**

```
// Project: TestingContinue
// Created: 2015-02-06

//*** Set window size and title ***
SetWindowSize(1024, 768, 0)
SetWindowTitle("Testing continue")

//*** Clear the screen ***
ClearScreen()

//*** Repeat until answer is 1 ***
repeat
   //*** Generate two numbers ***
   no1 = Random(0,20)
   no2 = Random(0,20)
   //*** If the second number is zero, skip ***
   //*** the remainder of this iteration    ***
   if no2 = 0
      Print("Skipping this iteration")
      Sync()
      Sleep(1000)
      continue
   endif
   //*** Calculate answer ***
   answer = no1 / no2
   //*** Display answer for 1 second ***
   PrintC(no1)
   PrintC(" / ")
   PrintC(no2)
   PrintC(" = ")
   Print(answer)
   Sync()
   Sleep(1000)
until answer = 1

do
   Print("Program complete")
   Sync()
loop
```

In the case of a **for** loop, executing the **continue** statement causes control to exit the current iteration but the loop counter is incremented before the next iteration is executed. The program in FIG-6.13 uses a **continue** statement within a **for** loop to display only the even numbers between 1 and 20.

FIG-6.13

Using **continue** in a **for** Loop

```
// Project: TestingContinue2
// Created: 2015-02-06


//*** Set window size and title ***
SetWindowSize(1024, 768, 0)
SetWindowTitle("Testing continue in a for loop")

//*** Clear the screen ***
ClearScreen()

do
   //*** For 20 times DO ***
   for c = 1 to 20
      //*** If it's an odd number, skip this iteration ***
      if Mod(c,2) <> 0
         continue
      endif
      //*** Display the number ***
      Print(c)
   next c
   Sync()
loop
```

✎

This approach isn't the best way to display even numbers but is used here only to demonstrate how the **continue** statement operates.

Activity 6.19

Create a new project called *TestingContinue2* and implement the code in FIG-6.13.

Test your program and check that only even numbers are displayed.

Modify the program, replacing the **continue** command with an **exit** command.
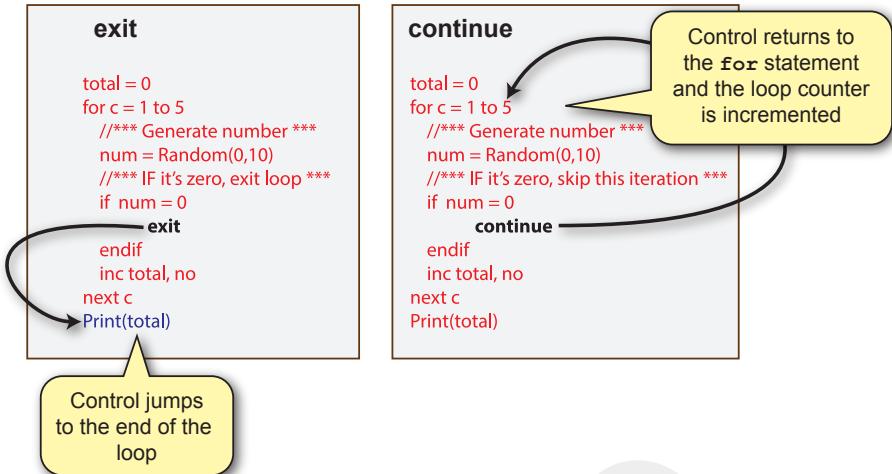
How does the new display differ from the original program's display?

The results from Activity 6.19 highlight the difference between **continue** and **exit** with the first terminating the current iteration only while the second terminates the whole looping operation.

FIG-6.14 shows the difference in the flow of control between these two commands.

FIG-6.14

Comparing **exit** and
continue

```
exit

total = 0
for c = 1 to 5
   //*** Generate number ***
   num = Random(0,10)
   //*** IF it's zero, exit loop ***
   if  num = 0
      exit
   endif
   inc total, no
next c
Print(total)
```

Control jumps
to the end of the
loop

```
continue

total = 0
for c = 1 to 5
   //*** Generate number ***
   num = Random(0,10)
   //*** IF it's zero, skip this iteration ***
   if  num = 0
      continue
   endif
   inc total, no
next c
Print(total)
```

Control returns to
the **for** statement
and the loop counter
is incremented

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.

Under normal circumstances, when a **do** loop is executing, 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.
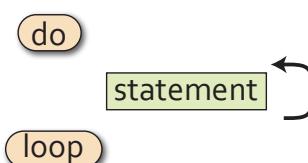
As we write more complex programs you will begin to understand why a **do** loop is so often needed to get the game to run smoothly.

Another reason that this infinite loop structure is useful is that apps designed to be run on tablets and smartphones very rarely close on their own, but keep running until closed by some external command.

The **do...loop** structure takes the format shown in FIG-6.15.

FIG-6.15

do...loop


```
do
   statement
loop
```

As we've seen in all of our previous programs, the main use of this structure is as an indefinite loop at the end of our logic.

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 6. 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 6.20

Turn the above algorithm into an AGK BASIC project, *AverageScore*, using the *Buttons* files to allow input.

Test the program, making sure it operates as expected.

Nested `for` Loops

Perhaps the commonest nested loops are nested `for` loops. And, although someone new to programming can sometimes have difficulties with the concept, it is 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 first two digits of the odometer. As you travel along you'll see the 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-6.16 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-6.16

Nested **for** loops

```
// Project: NestedFor
// Created: 2015-01-21

//*** Set window properties ***
SetWindowTitle("Nested For Loops")
SetWindowSize(1024,768,0)
```

FIG-6.16
(continued)

Nested **for** loops

```
SetDisplayAspect(1024.0/768)
ClearScreen()

do
   //*** Nested for loop ***
   for tens = 0 to 9    //Outer loop
      for units = 0 to 9   //Inner loop
         PrintC(tens)
         PrintC(" ")
         Print(units)
         Sync()
         Sleep(200)
      next units
   next tens
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 6.21

Start a new project, *NestedFor*, and code the program to match FIG-6.16.

Test your project.

Activity 6.22

What would be output by the following code?

```
for no1 = -2 to 1
    for no2 = 0 to 3
            PrintC(no1)
            PrintC(" ")
            Print(no2)
        next no2
    next no1
```
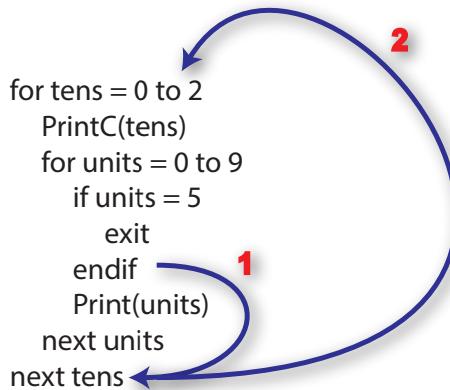
Nested Loops and the `exit` and `continue` Statements

Where we have a nested loop structure and an **exit** or **continue** statement is placed within the loop body of an inner loop, then control jumps to the end of that inner loop only, execution of the outer loop continues as normal.

The example in FIG-6.17 shows how an **exit** statement affects the flow of control when placed within an inner **for** loop.

FIG-6.17

```
for tens = 0 to 2
    PrintC(tens)
    for units = 0 to 9
        if units = 5
            exit
        endif
        Print(units)
    next units
next tens
```

1   When the **exit** statement is executed control jumps to the first statement after the end of the *units* **for** loop...

2   ...since this marks the end of the outer **for** loop (*units*), the program jumps back to the start of that **for** loop, increments *tens* and continues as normal.

Activity 6.23

Start a new project, *NestedJump*, and create a program which includes the code given in FIG-6.17.

What values are displayed when the program is run?

What values would be displayed if we replaced the term **exit** with the term **continue**?

Testing Iteration 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 the updated *Guess* program we added statements to ensure that the guess entered was in the range 1 to 100 using the following code:

```
guess = GetButtonEntry()
while guess < 1 or guess > 100
    Print("Your guess must be between 1 and 100")
    Print("Enter your guess again(1 - 100) : ")
    Sync()
```

```
        Sleep(2000)
        guess = GetButtonEntry()
    endwhile
```

To test the `while` loop in this code we could use the test data shown in FIG-6.18.

FIG-6.18

Test Data

Test No.	guess
1	23
2	101, 5
3	180, 121, 32

The `while` loop is only executed if *guess* is outside the range 1 to 100, 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 (101) 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 (180 and 121) before entering a valid value (32), causing the `while` loop body to execute twice.

Activity 6.24

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(1500)
        num = GetButtonEntry()
endwhile
average = total / count
do
        PrintC("Average is ")
        Print(average)
        Sync()
loop
```

Make up a set of test values (similar in construct to FIG-6.18) for the `while` loop in the code.

Create a new project, *AverageTest*, 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 therefore 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.

A `do` loop can only be tested for zero and one iteration if it contains an `exit` statement.

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` loop 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.

- The `continue` statement can be used to exit the current iteration of a loop.

- One loop structure can be placed within another loop structure. Such a

structure is known as a nested loop.

- When an `exit` or `continue` statement is placed within the inner loop of a nested loop structure, it is only that inner loop that is affected when control exits the loop structure (`exit`) or the current iteration (`continue`).

- Loops should be tested by creating test data for zero, one and multiple iterations during execution whenever possible.

Debugging

Introduction

Unfortunately, we all suffer from logic errors when creating programs which are more than a few pages in length. We may have coded a calculation incorrectly, used the wrong variable in an assignment, or tested for the wrong set of conditions. We may even have omitted a vital test or calculation.

To solve these problems, we need to use Sherlock-Holmes-type cunning to understand the clues our faulty results produce as an aid to finding out where in our code things are going wrong.

Detecting and correcting logic errors is known as **debugging**.

Using Extra Code

One way to discover where the problem lies is to add `Print` and/or `Message()` statements to our program. These can not only be used to display the contents of a variable, but can also let us know which part of our code is being executed. For example, in the code

```
if units = 5
    Message("Exiting loop")
    exit
```

the `Message()` statement is used to show we have entered the code we expect to be executed when the variable *units* contains the value 5 . If the message fails to appear at the correct time, we would know that there is some fault in the preceding `if` statement or in how *units* is assigned its value.

We might also want to discover the value held in a variable *total* while the program is running, and we could do this with the line

```
Print(total)
```

This might bring to light the fact that *total* was not being correctly added to.

Using a Debugger

A much better way to discover the logic errors in our code is to use a debugger. Typically, a debugger will allow a program to be executed one statement at a time (known as **single-stepping**) under user control or to execute a program as normal until it reaches a specific statement in the code and then to single step through the code.

Getting the program to stop execution at a specific statement is achieved by adding a **breakpoint** at that position in the code. If necessary, a program can contain many breakpoints.

When we single-step through a program, the debugger will highlight the source code line which is about to be executed. Typically, the next statement is executed by pressing one of the function keys.

As well as being able to control the execution of our program, a debugger will allow us to see the current value in any set of specified variables, with these values updating automatically as each line of the program is executed.

Debugging in AGK

To add a break point to an AGK program, all we have to do is click just to the right of the line number at which we wish the program to stop. In FIG-6.19, a break point has been added at line 41 in the project *AverageTest*. This means the program will halt just before the **endwhile** statement is executed.

FIG-6.19

Adding a Breakpoint

✎

To remove a breakpoint, just click on the same spot for a second time.

To discover the contents of a variable while the program is running, we need to enter the variable's name in the debug page of the Project Panel. In FIG-6.20, the variables *average*, *total* and *count* have been added.
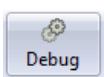
FIG-6.20

Watching Variables

Debug

If we are using the debugger, we need to click on the **Debug** button in the Tool bar rather than use the **Compile** and **Run** buttons.

Now the program will run until it reaches the first breakpoint. If we want to single step our way through the whole program, we'd place a breakpoint in the first line.

When the break point is reached, we can either:

- Press F9 to continue execution until another breakpoint is encountered.
- Press F10 to execute the next statement in the program.

If we've pressed F10, we need to continue pressing it for each statement we want executed, but at any time we may press F9 to execute up to the next breakpoint.

Activity 6.25

Reload *AverageTest* and add a breakpoint at the `endwhile` statement.

In the Project Panel, click on the *Debug* tag.

In the *Variables* table of the *Debug* panel, click twice under the *variable* column and enter the word *average* and press the *Enter* key.

Click twice under *average* and enter *total*. Under *total*, enter *count*.

Press the **Debug** button in the Tool bar.

Organise the IDE and *AverageTest* app windows so that both are visible.

In the running app, enter the numbers 12 then 3.

Now, click on the AGK IDE window to give it focus. Notice that the red circle of the breakpoint now contains a yellow arrow, pointing towards the code. This indicates the next line to be executed.

Notice also that in the *Debug* page on the left of the IDE, the variable *total* contains the value 12 and *count* 1.

Now press F10 to execute a single line of code (it's the `endwhile` that's executed and the yellow arrow moves back to the start of the `while` loop).

Press F10 for a second time. Now the `inc` line has the yellow arrow.

Press F10 again. This executes the `inc` line. Notice that the *Debug* page now shows *total* at 15. Press F10 again. The second `inc` statement is executed and *count* shows the value 2.

Now press F9 (which will run the program as normal until the breakpoint is reached) and enter the value zero.

Now press F10 repeatedly and watch as the condition in the `while` statement is found to be false and control jumps to `endwhile` and then to the line that calculates *average*. As more lines are executed, the average value will be displayed in the app. At this point, press the **End Debug** button in the IDE to terminate the program.

Debug

When pressing F9 or F10, the IDE must have focus, so remember to click in its window after entering data into the app.

When we are single-stepping in debug mode we have two choices on what should happen when we reach a statement that calls a user-defined function.
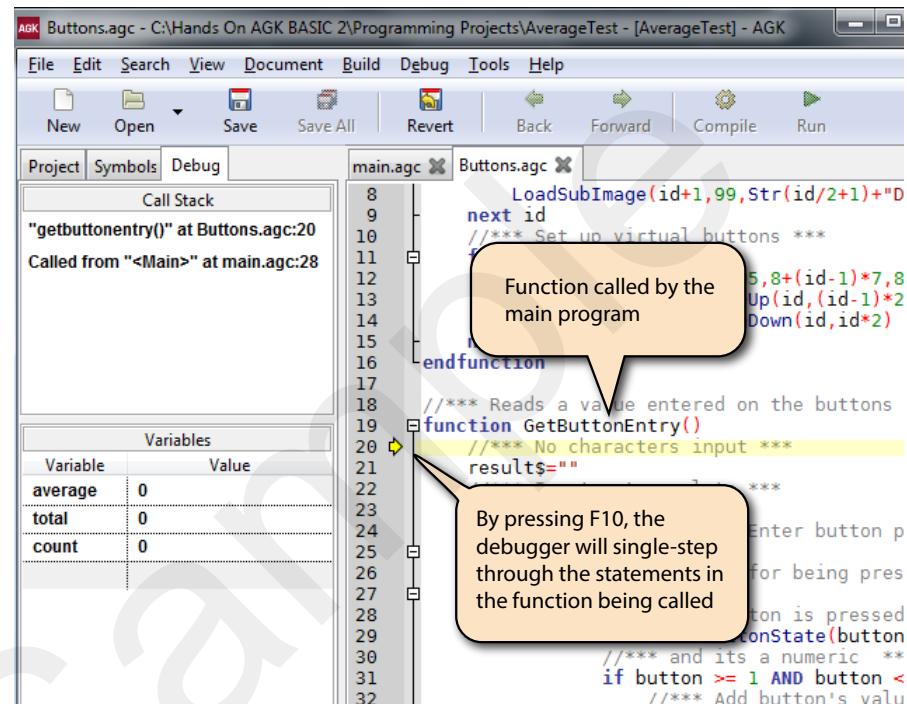
Although we will not be covering user-defined functions until Chapter 9, we already know that a function is little more than a block of code designed to perform a specific task. In our *AverageTest* program, we call the function *GetButtonEntry()* in the last statement of our `while` loop:

```
num = GetButtonEntry()
```

If we continue to press the F10 key on reaching this line, then the file containing the source code for the function will be loaded into a new tabbed page in the editor section of the IDE and single-stepping will proceed through the lines of the function (see FIG-6.21).

FIG-6.21

Single-Stepping into a Function

Activity 6.26

In *AverageTest*, remove the breakpoint from the `endwhile` line by clicking again to the right of the line number.

Add a new breakpoint at the line in which *GetButtonEntry()* is called just before the start of the `while` loop.

Press the **Debug** button to run the program and the F10 to execute the line containing the breakpoint. Observe how the code within the function is now displayed and its first line highlighted.
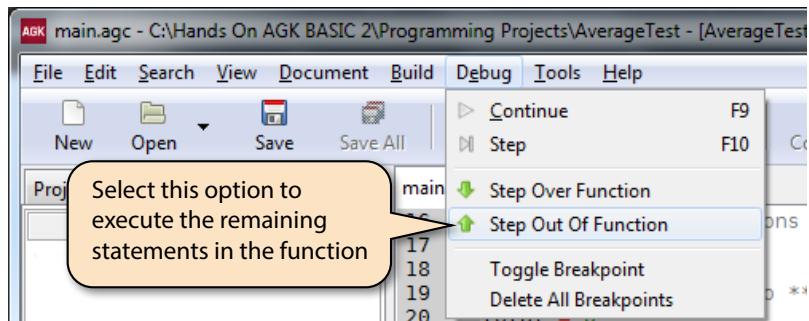
After single-stepping a few statements within the function, press the **End Debug** button in the Tools panel to terminate the debug session.

If we have accidentally single-stepped into a called function by pressing F10 too often, we can have the remainder of the function's code execute without the need for

single-stepping by selecting the **Debug|Step Out Of Function** option from the main menu (see FIG-6.22).

FIG-6.22

Executing the Remaining
Code in a Function

> **Activity 6.27**
>
> Re-run *AverageTest*, retaining the previous breakpoint.
>
> Press the **Debug** button to run the program and the F10 to execute the line containing the breakpoint.
>
> Once the yellow arrow highlighting the next line to be executed is within the code for the *GetButtonEntry()* function, select **Debug|Step Out Of Function** from the main menu. Notice that the remainder of the function's code has now been executed (and it is waiting for you to enter another number).
>
> Terminate program execution.

As a general rule, we won't want to single step our way through an existing function (which should have been fully tested previously during its creation). So, when we are single-stepping through a program and arrive at a function call, we can have the debugger execute the function's code as if it were a single statement by selecting the **Menu| Step Over Function** option from the main menu.

> **Activity 6.28**
>
> Re-run *AverageTest*, making sure it still has a breakpoint at the first call to *GetButtonEntry()*.
>
> Press the **Debug** button to run the program as far as the breakpoint.
>
> Now select **Menu| Step Over Function** from the main menu and observe how the function's code is executed as if were a single statement and how the yellow arrow highlighter moves onto the next line in the main program once you've entered a number on the button keys.
>
> Terminate program execution.

The Call Stack

Another part of the debugger is the **Call Stack** area in the Project Panel's *Debug* page. This area is used to tell which module and code line is about to be executed. In this context, the main program is identified as module **<Main>**. FIG-6.23 shows the contents of the *Call Stack* area when we have single-stepped the first two lines of *AverageTest*.

FIG-6.23

The Call Stack in
<Main>

When the line to be executed is within a function called by the main program, then this is detailed in *Call Stack* (see FIG-6.24).

FIG-6.24

The Call Stack in a
Function

The Debug Log

A final display related to debugging is found in the Report Panel at the bottom of the IDE window. Here there is a tagged page labelled **Debug Log**. Normally, this will only contain the standard debug message (see FIG-6.25) but it can also be used as the output are for debug messages generated by the program being tested.

FIG-6.25

The Debug Log

Log()

The `Log()` command acts rather like `Print()` but outputs to the *Debug Log* page rather than the app window. The statement has the format shown in FIG-6.26.

FIG-6.26

Log()

where:

message	is a string containing the text to be displayed on the *Debug Log* page.

For example, rather than single-step through a long program, we might wish to output messages to the *Debug Log* using statements such as

```
if no < 0
    Print("Negative")
else
    Log("no < 0 is false")
        ...
```

The output from `Log()` only appears when running an app in Debug mode. During a normal run, the statement is ignored.

Debugging While Running the App on Another Device
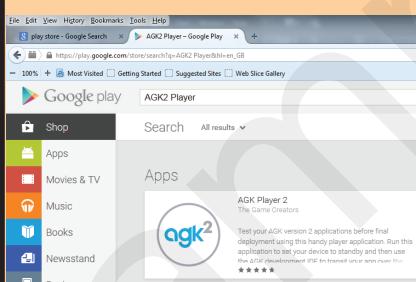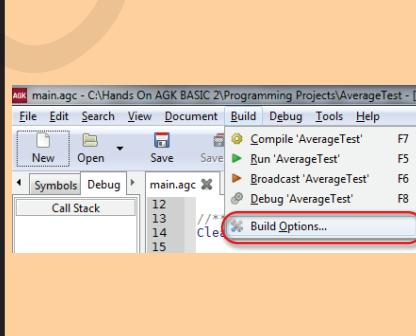
Although the default is to have the app run on our desktop, we can also use the debug features when running on a mobile device. To do this we need to perform the sequence of events shown in FIG-6.27.

FIG-6.27

Debugging An App Running on a Mobile Device

✎

To run the app on a mobile device, there must be a WiFi connection to the desktop.

To use a mobile device to run your app during a debug session we must have a version of the *AGK Player 2* created after March 2015.	If we run the player on our mobile device, we'll see its IP address. This should be a number such as 192.168. 0.9
Returning to AGK's IDE, we now need to select **Build|Build Options...** from the main menu.	In the dialog box that appears, we need to enter the IP address of the mobile device in the bottom field of the Build page.

With these details set up, all we need to do is make sure the *AGK Player 2* is running on the mobile device then hit the debug button (or press F8). Now the app will run on the new device while we continue to see the code and debug details on the desktop.

Solutions

Activity 6.1

Modified code for *Guess*:

```
// Project: Guess
// Created: 2015-01-011

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Guess")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Generate number (0 to 9) ***
number = Random(0,9)

//*** Display user prompt ***
Print("Guess what my number was (0 to 9) : ")
Sync()
Sleep(2000)

//*** Get a guess in range 0 to 9 ***
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

//*** Calculate difference ***
diff = number - guess

do
  //*** Respond to 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

  //*** Display number generated  ***
  PrintC("My number was : ")
  Print(number)
  PrintC("Your guess was : ")
  Print(guess)
  Sync()
loop
```

Activity 6.2

Code for *DiceCount*:

```
// Project: DiceCount
// Created: 2015-01-12

//*** Set window properties ***
SetWindowTitle("Dice Count")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Set count to zero ***
count = 0

//*** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)

//*** display dice values ***
PrintC(dice1)
PrintC(" ")
Print(dice2)
Sync()
Sleep(500)

//*** Keep going while total is less than 9 ***
while dice1 + dice2 <= 8
  //*** add 1 to count ***
  count = count + 1
  //*** Throw dice ***
  dice1 = Random(1,6)
  dice2 = Random(1,6)
  //*** Display dice values ***
  PrintC(dice1)
  PrintC(" ")
  Print(dice2)
  Sync()
  Sleep(500)
endwhile

//*** Display final result ***
do
  PrintC("You had a run of ")
  PrintC(count)
  Print(" throws")
  Sync()
loop
```

Activity 6.3

Code for *Total*:

```
// Project: Total
// Created: 2015-01-12

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Total")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Set up buttons ***
SetUpButtons()

//*** Set total to zero ***
total = 0

//*** Keep going until zero entered ***
repeat
  //*** Get value ***
  no = GetButtonEntry()
  //*** Add value to total ***
  total = total + no
until no = 0

do
  //*** Display total ***
  PrintC("Total = ")
  Print(total)
  Sync()
loop
```

Activity 6.4

Modified code for *Guess* (remember to indent all the code between the `repeat` and `until` terms):

```
// Project: Guess
// Created: 2015-01-011

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Guess")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()
```

```
//*** Generate number (0 to 9) ***
number = Random(0,9)

//*** Display user prompt ***
Print("Guess what my number is (0 to 9) : ")
Sync()
Sleep(2000)

//*** Keep guessing until correct ***
repeat
   //*** Get a guess in range 0 to 9 ***
   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
   //*** Calculate difference ***
   diff = number - guess
   //*** Respond to 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
   Sync()
   Sleep(2000)
until guess = number

do
   //*** Display number generated  ***
   PrintC("My number was : ")
   Print(number)
   Sync()
loop
```

Notice that the nested `if` structure has been moved outside the `do...loop` and that the final display no longer prints the *guess* value (since this must be the same as *number* at this point).

Activity 6.5

```
for j = 1 to 10
```

Activity 6.6

This code would display the values 1 to 10.

Activity 6.7

Version of *Tables* for the 12 times table:

```
// Project: Tables
// Created: 2015-01-22

//*** Set window title and size ***
SetWindowTitle("Tables")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

do
   for c = 12 to 144 step 12
      Print(c)
   next c
   Sync()
loop
```

Activity 6.8

Modified version of *Tables*:

```
// Project: Tables
// Created: 2015-01-22

//*** Set window title and size ***
SetWindowTitle("Tables")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

do
   for c = 144 to 12 step -12
      Print(c)
   next c
   Sync()
loop
```

Activity 6.9

Modified code for *UserLoop* (lower limit):

```
// Project: UserLoop
// Created: 2015-01-21

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window title and size ***
SetWindowTitle("User Loop")
SetWindowSize(720,1024,0)
SetDisplayAspect(720.0/1024)

//*** Clear the screen ***
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Get lower limit ***
Print("Enter for loop lower limit : ")
Sync()
Sleep(2000)
//*** Get lower limit ***
low = GetButtonEntry()

//*** Get upper limit ***
Print("Enter for loop upper limit : ")
Sync()
Sleep(2000)
high = GetButtonEntry()

//*** Display values low to high ***
do
   for c = low to high
      Print(c)
   next c
   Sync()
loop
```

Modified code for *UserLoop* (step size):

```
// Project: UserLoop
// Created: 2015-01-21

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window title and size ***
SetWindowTitle("User Loop")
SetWindowSize(720,1024,0)
SetDisplayAspect(720.0/1024)

//*** Clear the screen ***
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Get lower limit ***
Print("Enter for loop lower limit : ")
Sync()
Sleep(2000)
//*** Get lower limit ***
low = GetButtonEntry()
```

```
//*** Get upper limit ***
Print("Enter for loop upper limit : ")
Sync()
Sleep(2000)
high = GetButtonEntry()

//*** Get step size ***
Print("Enter for loop step size : ")
Sync()
Sleep(2000)
increment = GetButtonEntry()

//*** Display values 1 to high ***
do
   for c = low to high step increment
      Print(c)
   next c
   Sync()
loop
```

Activity 6.10

Code for *ForReal*:

```
// Project: ForReal
// Created: 2015-01-22

//*** Set window title and size ***
SetWindowTitle("For Real")
SetWindowSize(720,1024,0)
SetDisplayAspect(720.0/1024)

//*** Clear the screen ***
ClearScreen()

do
   for c# = 1.0 to 2.0 step 0.1
      Print(c#)
   next c#
   Sync()
loop
```

Notice that the values displayed are 1.0 to 1.9. 2.0 does not appear.

Activity 6.11

Modified version of *ForReal*:

```
// Project: ForReal
// Created: 2015-01-22

//*** Set window title and size ***
SetWindowTitle("For Real")
SetWindowSize(720,1024,0)
SetDisplayAspect(720.0/1024)

//*** Clear the screen ***
ClearScreen()

do
   for c# = 1.0 to 2.0 step 0.25
      Print(c#)
   next c#
   Sync()
loop
```

The display now runs from 1.0 to 2.0 (in steps of 0.25).

Activity 6.12

Code for *InTotal*:

```
// Project: InTotal
// Created: 2015-01-22

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window title and size ***
SetWindowTitle("In Total")
SetWindowSize(720,1024,0)
SetDisplayAspect(720.0/1024)

//*** Clear the screen ***
ClearScreen()
```

```
//*** Display the buttons ***
SetUpButtons()

//*** Set total to zero ***
total = 0

//*** Read in and total six values ***
for c = 1 to 6
   PrintC("Enter number ")
   PrintC(c)
   Print(" : ")
   Sync()
   Sleep(1500)
   num = GetButtonEntry()
   inc total, num
next c

//*** Display total ***
do
   PrintC("Total is ")
   Print(total)
   Sync()
loop
```

Activity 6.13

Code for *Shades*:

```
// Project: Shades
// Created: 2015-01-22

//*** Set window title and size ***
SetWindowTitle("Shades")
SetWindowSize(720,1024,0)
SetDisplayAspect(720.0/1024)

//*** Clear the screen ***
ClearScreen()

//*** Cyscle through all shades of red ***
do
   for red = 0 to 255
      SetClearColor(red,0,0)
      Sync()
      Sleep(25)
   next red
loop
```

Activity 6.14

Code for *SmallestNumber*:

```
// Project: SmallestNumber
// Created: 2015-01-22

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Smallest Number")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Get number ***
Print("Enter number ")
Sync()
Sleep(1500)
no = GetButtonEntry()

//*** Set smallest to first number ***
smallest = no

//*** For 4 times do ***
for c = 1 to 4
   //*** Get next number ***
   Print("Enter number ")
   Sync()
   Sleep(1500)
   no = GetButtonEntry()
   //*** If number smaller, record it ***
   if no < smallest
      smallest = no
   endif
next c
```

```
do
   //*** Display smallest value ***
   PrintC("Smallest value entered was ")
   Print(smallest)
   Sync()
loop
```

Modified version of *SmallestNumber*:

```
// Project: SmallestNumber
// Created: 2015-01-22

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Largest Number")
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)

//*** Clear the screen ***
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Get number ***
Print("Enter number ")
Sync()
Sleep(1500)
no = GetButtonEntry()

//*** Set largest to first number ***
largest = no

//*** For 4 times do ***
for c = 1 to 4
    //*** Get next number ***
    Print("Enter number ")
    Sync()
    Sleep(1500)
    no = GetButtonEntry()
    //*** If number larger, record it ***
    if no > largest
        largest = no
    endif
next c

do
   //*** Display largest value ***
   PrintC("Largest value entered was ")
   Print(largest)
   Sync()
loop
```

Of course, the project name is really no longer appropriate!

Activity 6.15

Modified version of *SumDice*:

```
// Project: SumDice
// Created: 2015-01-21

//*** Set window properties ***
SetWindowTitle("Sum Dice")
SetWindowSize(1024,768,0)
SetDisplayAspect(1024.0/768)
ClearScreen()

//*** Set total to zero ***
total = 0
//*** for 5 times do ***
for c = 1 to 5
   //*** Display number of rolls so far ***
   PrintC("Roll number ")
   Print(c)
   Sync()
   Sleep(1500)
   //*** Throw both dice ***
   dice1 = Random(1,6)
   dice2 = Random(1,6)
   //*** Display throw number and dice values ***
   PrintC("dice 1 : ")
   PrintC(dice1)
   PrintC("          dice 2 : ")
   Print(dice2)
   Sync()
   Sleep(4000)
```

```
   //*** if either dice is a 1 then quit loop ***
   if dice1 = 1 or dice2 = 1
      exit
   endif
   //*** Add dice throws to total ***
   total = total + dice1 + dice2
next c

do
   //*** Display final score ***
   PrintC("Your final score was : ")
   Print(total)
   PrintC("After ")
   PrintC(c)
   Print(" throws")
   Sync()
loop
```

When no 1 is thrown, the program displays the message *After 6 throws*. However, only five throws have been made.

Activity 6.16

Modified version of *SumDice*:

```
// Project: SumDice
// Created: 2015-01-21

//*** Set window properties ***
SetWindowTitle("Sum Dice")
SetWindowSize(1024,768,0)
SetDisplayAspect(1024.0/768)
ClearScreen()

//*** Set total to zero ***
total = 0
//*** for 5 times do ***
for c = 1 to 5
   //*** Display number of rolls so far ***
   PrintC("Roll number ")
   Print(c)
   Sync()
   Sleep(1500)
   //*** Throw both dice ***
   dice1 = Random(1,6)
   dice2 = Random(1,6)
   //*** Display throw number and dice values ***
   PrintC("dice 1 : ")
   PrintC(dice1)
   PrintC("          dice 2 : ")
   Print(dice2)
   Sync()
   Sleep(4000)
   //*** if either dice is a 1 then quit loop ***
   if dice1 = 1 or dice2 = 1
      exit
   endif
   //*** Add dice throws to total ***
   total = total + dice1 + dice2
next c

//*** Adjust c if no 1s thrown ***
if c = 6
   dec c
endif

do
   //*** Display final score ***
   PrintC("Your final score was : ")
   Print(total)
   PrintC("After ")
   PrintC(c)
   Print(" throws")
   Sync()
loop
```

Activity 6.17

Modified code for *Guess*:

```
// Project: Guess
// Created: 2015-01-011

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window properties ***
SetWindowTitle("Guess")
```

```
SetWindowSize(768, 1024, 0)
SetDisplayAspect(768/1024.0)
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Generate number (1 to 100) ***
number = Random(1,100)

//*** Display user prompt ***
Print("Guess what my number is (1 to 100) : ")
Sync()
Sleep(2000)

//*** Allow up to seven guesses ***
for c = 1 to 7
  //*** Get a guess in range 1 to 100 ***
  guess = GetButtonEntry()
  while guess < 1 or guess > 100
    Print("Your guess must be between 1 and 100")
    Print("Enter your guess again(1 - 100) : ")
    Sync()
    Sleep(2000)
    guess = GetButtonEntry()
  endwhile
  //*** Calculate difference ***
  diff = number - guess
  //*** Respond to 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")
        exit
      else
        if diff >= -2
          Print("Your guess is slightly too high")
        else
          Print("Your guess is too high")
        endif
      endif
    endif
  endif
  Sync()
  Sleep(2000)
next c

do
  //*** Display number generated and number of
  ⤷guesses ***
  PrintC("My number was : ")
  Print(number)
  if c = 8
    Print("You failed to guess the number ")
  else
    PrintC("It took you ")
    PrintC(c)
    Print(" guesses")
  endif
  Sync()
loop
```

Activity 6.18

No solution required.

Activity 6.19

When an `exit` command is used, no output appears. This is because the first value of c (1) is not even and so the `exit` command is executed and looping terminated.

Activity 6.20

Code for *AverageScore*:

```
// Project: AverageScore
// Created: 2015-01-22

//*** Other source file used by program ***
#include "Buttons.agc"
```

```
//*** Set window title and size ***
SetWindowTitle("Average Score")
SetWindowSize(720,1024,0)
SetDisplayAspect(720.0/1024)

//*** Clear the screen ***
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Set total to zero ***
total = 0

//*** for 6 times do ***
for c = 1 to 6
  //*** Read score ***
  PrintC("Enter score ")
  Print(c)
  Sync()
  Sleep(1500)
  score = GetButtonEntry()
  //*** while score is invalid do ***
  while score < 0 or score > 100
    //*** Display error message ***
    Print("Score must be between 0 to 100")
    Sync()
    Sleep(1500)
    //*** Get score ***
    PrintC("Enter score ")
    Print(c)
    Sync()
    Sleep(1500)
    score = GetButtonEntry()
  endwhile
  //*** Add score to total ***
  inc total, score
next c

//*** Calculate average score ***
average# = total/6.0

//*** display average score ***
do
  PrintC("Average score is ")
  Print(average#)
  Sync()
loop
```

Activity 6.21

No solution required.

Activity 6.22

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 display between each set of values.

Activity 6.23

Code for *NestedJump*:

```
// Project: NestedJump
// Created: 2015-02-06
```

```
//*** Set window size and title ***
SetWindowSize(1024, 768, 0)
SetWindowTitle("Nested Jump")

//*** Clear the screen ***
ClearScreen()

do
  for tens = 0 to 2     //Outer loop
    PrintC(tens)
    for units = 0 to 9 //Inner loop
      if units = 5
        exit
      endif
      Print(units)
    next units
  next tens
  Sync()
loop
```

The program will create the following display:

```
00
1
2
3
4
11
2
3
4
21
2
3
4
```

As soon as the inner loop (*units*) reaches 5, the loop exits, so the values 5 to 9 are never displayed.

When `continue` is used in place of `exit`, all combinations from 00 to 29 are displayed except for 05, 15 and 25.

Activity 6.24

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:

	num	Expected Results (for *average*)
Test 1	0	0
Test 2	8,0	8
Test 3	12,7,0	9.5

Code for *AverageTest*:

```
// Project: AverageTest
// Created: 2015-01-22

//*** Other source file used by program ***
#include "Buttons.agc"

//*** Set window title and size ***
SetWindowTitle("Average Test")
SetWindowSize(720,1024,0)
SetDisplayAspect(720.0/1024)

//*** Clear the screen ***
ClearScreen()

//*** Display the buttons ***
SetUpButtons()

//*** Set total to zero ***
total = 0
//*** Set count to zero ***
count = 0

//*** Read number ***
```

```
Print("Enter number (0 to stop)")
Sync()
Sleep(1500)
num = GetButtonEntry()

//*** while num not zero do ***
while num <> 0
  //*** Add num to total ***
  inc total, num
  //*** Increment count ***
  inc count
  //*** Get next number ***
  Print("Enter number (0 to stop)")
  Sync()
  Sleep(1500)
  num = GetButtonEntry()
endwhile

//*** Calculate average ***
average# = total / count

do
  PrintC("Average is ")
  Print(average#)
  Sync()
loop
```

When we run the program with the test data, it turns out that the first run halts the program!

The line

```
average# = total/count
```

causes the program to crash. This is because *count* would have the value zero and hence the calculation would cause a division by zero error.

We can solve the problem by changing the code to

```
if count = 0
  average# = 0
else
  average# = total / count
endif
```

The third test given here would also cause a problem – giving a result of 9 rather than 9.5.

Since *total* and *count* are both integer variables, we get an integer result when calculating *average#*.

To solve this we need to use a float variable for either total or count. One possible solution is to write

```
count# = count
average# = total/count#
```

Activity 6.25

No solution required.

Activity 6.26

No solution required.

Activity 6.27

No solution required.

Activity 6.28

No solution required.

Activity 6.29

No solution required.

7

A First Look at Resources

In this Chapter:

❑ **The Screen Coordinates System**

❑ **Drawing Commands**

❑ **Introducing Images**

❑ **Introducing Sprites**

❑ **Introducing User Interaction**

❑ **Introducing Text**

Drawing Functions

Introduction

So far all we've done in the way of output is to create a little text. Not very eye-catching! But in this chapter we'll start to use some of those functions that allow us to draw basic shapes, display images, and create movement as well as have the user interact with those elements. But before we get started on those topics, we need to take a closer look at the screen and how we specify positions on that screen.

The Screen Coordinate System

If we want to draw shapes on the screen, then we need to specify where on the screen these elements are to appear. To do that we must take a closer look at the screen coordinate system we last covered in Chapter 3.
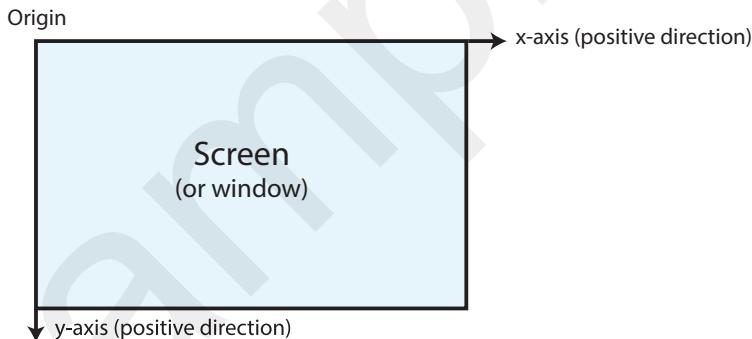
If you are unfamiliar with the maths behind a 2D coordinate system, you might like to read the *10 Minute Maths: Cartesian Coordinates* booklet available from Amazon.

✎

If you are running the app within a window, then its the top left of that window that is taken as the origin.

For those of us used only to maths coordinate systems, what's strange about computer screen coordinates is that the origin (point (0,0)) is at the top left corner of the screen and that the positive direction of the y-axis points downward (see FIG-7.1).

FIG-7.1

Screen Axes

Origin

Screen
(or window)

x-axis (positive direction)

y-axis (positive direction)
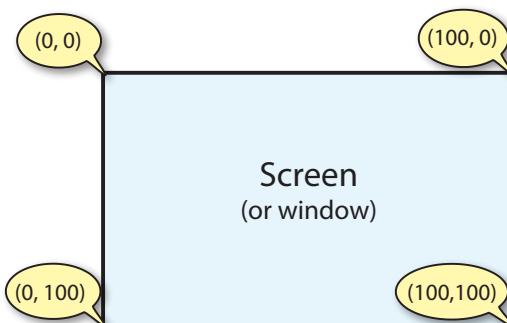
Exactly how we specify a point on the screen depends on the coordinate system we are using in our program,

If we are using the default percentage system, then the *x*-coordinate of any point on the screen lies between 0 and 100, with the *y*-coordinate also lying in the same range (see FIG-7.2).

FIG-7.2

The Percentage System

(0, 0)   (100, 0)

Screen
(or window)

(0, 100)   (100,100)
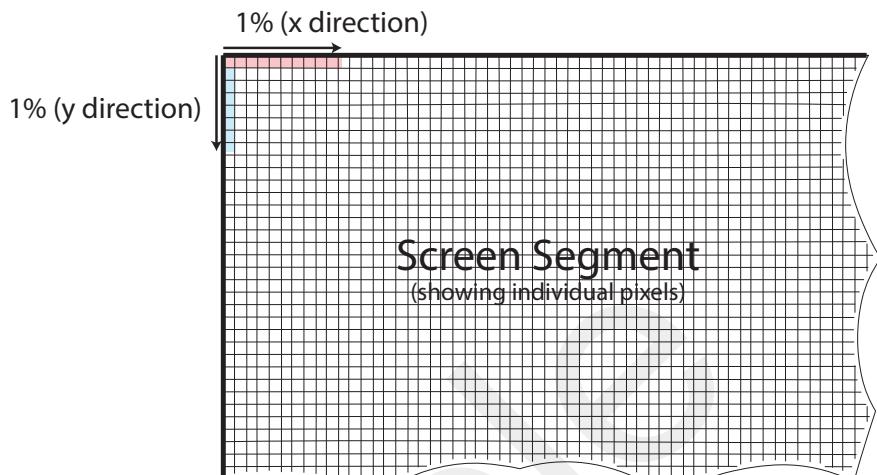
Note that, unless we have a square screen (or window), this means that the 1% along

the x-axis is not the same physical distance as 1% along the y-axis.

For example, let's say we are running our program in a window which is 1024 pixels wide by 768 pixels high (the same as the original iPad in landscape mode), then 1% in the *x* direction would cover 10.24 pixels while 1% in the *y* direction would only be 7.68 pixels (see FIG-7.3).

FIG-7.3

Percentage and Pixels

1% (x direction)

1% (y direction)

Screen Segment
(showing individual pixels)

And, since we can't change part of a pixel, the screen handler will round these figures to the nearest pixel meaning 1% in the *x* direction becomes10 pixels while 1% in the *y* direction is 8 pixels.

If we are employing the virtual pixels setup (using `SetVirtualResolution()` – see Chapter 3) the *x* and *y* distances will be specified in pixels.
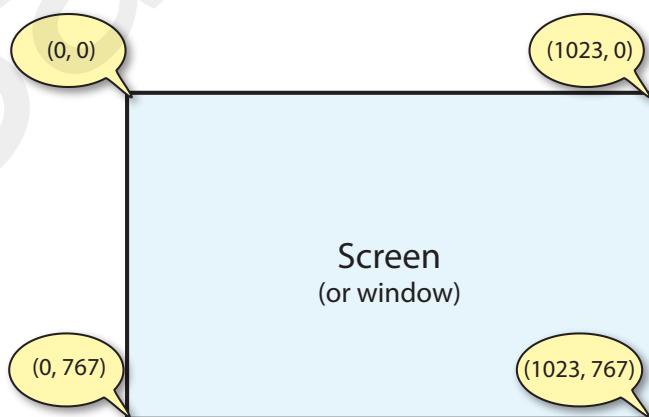
For example, if near the start of our program we have written

```
SetVirtualResolution(1024, 768)
```

then our on-screen *x*-axis would use measurements 0 to 1023 and the *y*-axis 0 to 767 (see FIG-7.4).

FIG-7.4

The Virtual Pixels System

(0, 0)   (1023, 0)

Screen
(or window)

(0, 767)   (1023, 767)

If we have set the virtual resolution so that it exactly matches the physical resolution of our screen or window, then we have the ideal situation where one virtual pixel is equivalent to one physical pixel. And if we intend to run our app on a single device, this is the way to go since we will have absolute precision when placing elements on the screen.

However, if we are designing an app which we expect to run on many different devices, then our virtual resolution is not going to be an exact match in all cases. For example, if we have set a virtual resolution of 1024 by 768 and then run the app on an iPad Air 2 with its screen resolution of 2048 by 1536, then 1 virtual pixel will be equivalent to two physical pixels.

Determining the Window and Screen Sizes

Screen Size

We can find out the size of our screen (in pixels) using the following two commands.

GetMaxDeviceWidth()

The width of the screen on which the AGK app is currently running can be found using the `GetMaxDeviceWidth()` function which has the format shown in FIG-7.5.

FIG-7.5

GetMaxDeviceWidth()

integer GetMaxDeviceWidth ( )

GetMaxDeviceHeight()

The height of the screen on which the AGK app is currently running can be found using the `GetMaxDeviceHeight()` function which has the format shown in FIG-7.6.

FIG-7.6

GetMaxDeviceHeight()

integer GetMaxDeviceHeight ( )

The program in FIG-7.7 displays the size of the current hardware's screen.

FIG-7.7

Finding the Screen's Dimensions

```
// Project: ScreenSize
// Created: 2015-01-23


//*** Window title and size ***
SetWindowTitle("Screen Size")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Get screen dimensions ***
screenwidth = GetMaxDeviceWidth()
screenheight = GetMaxDeviceHeight()

do
    //*** Display the screen's dimensions ***
    PrintC("Screen width: ")
    PrintC(screenwidth)
    PrintC(" pixels      Screen height: ")
    PrintC(screenheight)
    Print(" pixels")
    Sync()
loop
```

Window Size

When running our app within a window on a desktop machine, we can discover the window's dimensions using the following functions:

GetDeviceWidth()

This function returns the window's width in pixels and has the format shown in FIG-7.8.

integer (GetDeviceWidth) (( ))

GetDeviceHeight()

This function returns the window's height in pixels and has the format shown in FIG-7.9.

integer (GetDeviceHeight) (( ))

When these two commands are executed on a mobile device, the full screen dimensions are returned. That is to say, the functions return the same results as `GetMaxDeviceWidth()` and `GetMaxDeviceHeight()`.

Calculating the Percentage to Pixel Ratio

Early in this chapter we saw that, when using the percentage system, a 1% measurement in the *x* direction can cover a different number of pixels than a 1% measurement in the *y* direction.

To determine exactly how many pixels a distance of 1% covers in each direction we can use the following code:

```
one_percent_x# = GetDeviceWidth()/100.0
one_percent_y# = GetDeviceHeight()/100.0
```

Alternatively, to discover what percentage a single pixel represents, we can use

```
xpixel# = 100.0/GetDeviceWidth()
ypixel# = 100.0/GetDeviceHeight()
```

Remember to use 100.0 and not 100 in these calculations otherwise the program will perform integer division giving incorrect results.

> **Activity 7.3**
>
> Start a new project called *PercentPixel* and create a program to display the pixel-to-percent and percent-to-pixel values.
>
> Run the program on your desktop (with a window size of your choice) then run it again on a tablet or smartphone (by making use of *AGK Player 2*).

Defining Colour

MakeColor()

Although we often define a colour by supplying three separate integer values for its red, green and blue components, it is also possible to set up a single integer which contains all three colour values.

This is achieved using the `MakeColor()` statement whose syntax is shown in FIG-7.10.

FIG-7.10

MakeColor()

integer (MakeColor) (( ) [red] (,) [green] (,) [blue] ( ))

where

    red   is an integer value (0 to 255) giving the value of the red component of the desired colour.

    green   is an integer value (0 to 255) giving the value of the green component of the desired colour.

    blue   is an integer value (0 to 255) giving the value of the blue component of the desired colour.

The value returned by the statement will normally be stored in an integer variable for use in one of the other drawing statements.

We could create an integer value representing yellow with the line:

```
yellow = MakeColor(255,255,0)
```

Or we could create a random colour using:

```
unknown_colour =
↳MakeColor(Random(0,255),Random(0,255),Random(0,255))
```

GetColorRed(), GetColorGreen() and GetColorBlue()

We can discover the red, green and blue settings within a colour value set up by a previous call to `MakeColor()` using the statements `GetColorRed()`, `GetColorGreen()` and `GetColorBlue()` (see FIG-7.11).

FIG-7.11

GetColorRed()
GetColorGreen()
GetColorBlue()

where

col is an integer value returned by a previous call to `MakeColor()`.

The value returned by these statement is the integer value of the colour component and will lie in the range 0 to 255. For example, the code

```
colour = MakeColor(Random(0,255),Random(0,255),Random(0,255))
greenvalue = GetColorGreen(colour)
```

assigns the value of the green component within the variable *colour* to the variable *greenvalue*.

Drawing

AGK supplies three functions which allow us to draw basic shapes directly onto the screen. All of these draw functions have a property similar to the `Print()` statement in that they must be executed for each frame in order to remain visible.

DrawLine()

The `DrawLine()` function allows us to draw a straight line between two points.

The `DrawLine()` command can be employed in two different ways depending on which of the statement's formats is used. It can either draw a line of a single, specified colour or it can draw a line which gradually changes colour from one end of the line to the other. The formats of this statement are shown in FIG-7.12.

FIG-7.12

DrawLine()

Format 1

Format 2

where

x1, **y1** are real numbers giving the coordinates of the starting point of the line.

x2, **y2** are real numbers giving the coordinates of the finishing point of the line.

r, **g**, **b** (format 1) are integer values giving the red, green and blue components of the line's colour.

col1, col2 (format 2) are integer values for the colours at the start and end points of the line respectively. These colours will have been previously set up using `MakeColor()`.

The following program draws a single yellow-coloured line from the top-left corner to the bottom right (see FIG-7.13).

```
// Project: TestDrawLine
// Created: 2015-01-14

//*** Window title and size ***
SetWindowTitle("Test DrawLine")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

do
    //*** Draw a line from top-left to bottom-right ***
    DrawLine(0,0,100,100,255,255,0)
    Sync()
loop
```

Activity 7.4

Start a new project called *TestDrawLine* and implement and run the code given in FIG-7.13.

Modify the program to make use of two colour values, making the line change from red at the top left corner to yellow at the bottom right.

DrawBox()

The second drawing command available is `DrawBox()`. This draws a rectangle when supplied with the coordinates of the top-right and bottom-left corners.

The more unusual feature of the command is that it is possible to define a colour for each of the four corners of the rectangle which merge into each other in the displayed box. The box can also be solid (filled) or border-only (unfilled).

The format for the command is given in FIG-7.14.

FIG-7.14   DrawBox()

where

x1, **y1**	are real numbers giving the coordinates of the top-left corner of the required rectangle.
x2, **y2**	are real numbers giving the coordinates of the bottom-right corner.
col1, col2,col3, col4	are integer values giving the colours to be used in the four corners of the rectangle.
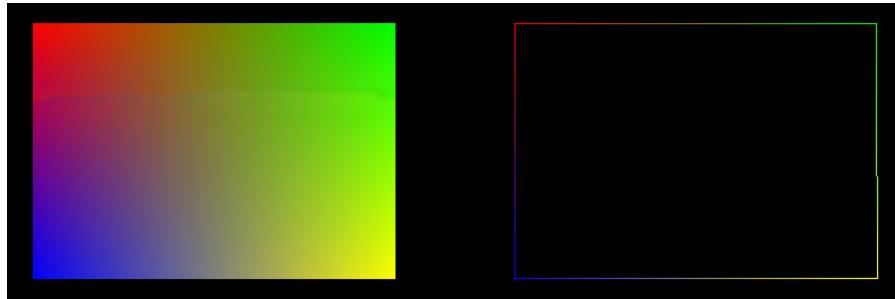
fill	is an integer value (0 or 1) which controls how the rectangle is drawn. 0: outline; 1: solid.

The colours are used to fill the box or, in the case of an outlined box, to colour the border lines (see FIG-7.15).

FIG-7.15

A Box

The program in FIG-7.16 makes use of the `DrawBox()` statement to draw a filled box.

FIG-7.16

Drawing a Box

```
// Project: Rectangles
// Created: 2015-01-14

//*** Window title and size ***
SetWindowTitle("Rectangles")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Create colour values ***
red = MakeColor(255,0,0)
green = MakeColor(0,255,0)
blue = MakeColor(0,0,255)
yellow = MakeColor(255,255,0)

do
    //*** Draw rectangle ***
    DrawBox(10,10,90,90,red,green,blue,yellow,1)
    Sync()
loop
```

Activity 7.5

Start a project called *Rectangles* implementing the code shown in FIG-7.16.

Run the program and observe the effect created.

Modify the program so that the four colours are selected randomly on each refresh of the screen.

Add a `Sleep()` statement to cause each colour combination to remain on screen for 500 milliseconds.

Run the program and observe the effect of the changing colours.
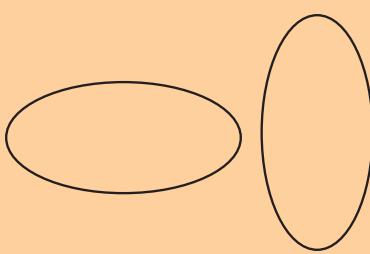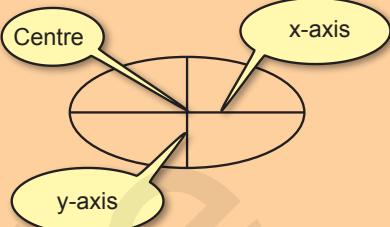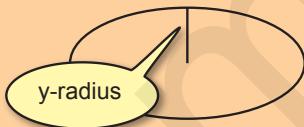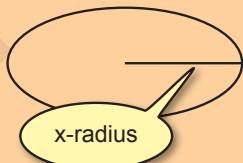
Modify the program again so that only the outline of the rectangle is drawn.

DrawEllipse()

We can think of an ellipse as a circle which has been stretched in one direction. A typical ellipse and its main properties are shown in FIG-7.17.

FIG-7.17

Characteristics of an Ellipse

An ellipse looks like a squashed circle.	It has a **centre**, an **x-axis** (along the x direction) and a **y-axis** (along the y direction).
	Centre, x-axis, y-axis
The vertical distance from the centre to the edge of the ellipse is its **y-radius**.	The horizontal distance from the centre to the edge of the ellipse is its **x-radius**.
y-radius	x-radius

To draw an ellipse within AGK we can use the `DrawEllipse()` statement (see FIG-7.18). The ellipse can be solid or in outline only with the specified two colours merging over the area of the ellipse.

FIG-7.18 DrawEllipse()

DrawEllipse (x , y , radx , rady , col1 , col2 , fill)

where

x,y	are real numbers giving the coordinates of the centre point of the ellipse.
radx	is a real number giving the length of the *x* radius.
rady	is a real number giving the length of the *y* radius.
col1, col2	are integer values giving the fill (or outline) colours for the ellipse.
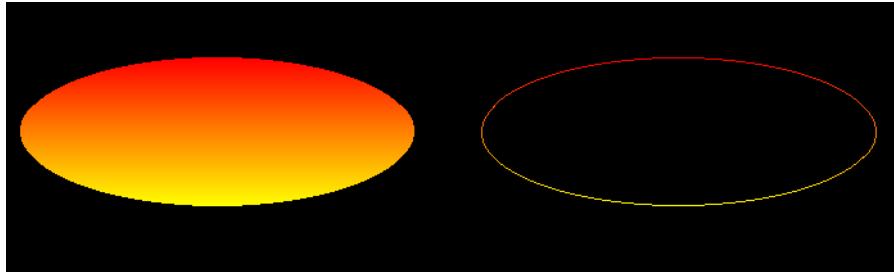
fill	is an integer value (0 or 1) which controls how the ellipse is drawn. 0: outline; 1: solid.

In a filled ellipse, *col1* defines the colour at the top of the ellipse, *col2* the colour at the bottom of the ellipse with the two gradually merging at the centre. For an outlined ellipse, the colours are used on the outline at the top (*col1*) and bottom (*col2*) parts of the outline (see FIG-7.19).

FIG-7.19

An Ellipse

The program in FIG-7.20 draws a filled ellipse which changes size and colour randomly once every second.

FIG-7.20

Drawing an Ellipse

```
// Project: Ellipses
// Created: 2015-01-15

//*** Set window title and size ***
SetWindowTitle( "Ellipses" )
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

do
    //*** Draw ellipse ***
    DrawEllipse(50,50,30,10,MakeColor(Random(0,255),Random
    ↳(0,255), Random(0,255)),MakeColor(Random(0,255),Random(0,255),
    ↳Random(0,255)),1)
    Sync()
    Sleep(500)
loop
```

> **Activity 7.6**
>
> Start a new project called *Ellipses* which implements the code shown in FIG-7.20.
>
> Change the program to draw a circle (with an x-radius of 20) instead of an ellipse.
>
> Test your program.

✎ Calculating the required *y*-radius in order to create a circle makes use of formulas previously used in Activity 7.3.

Summary

- Screen coordinates default to a percentage system with width and height each 100% irrespective of the actual window size used.

- Virtual pixels are an alternative to percentage coordinates and specify a screen size in virtual pixels which need not match the actual pixel size of the window used.

- Use `GetMaxDeviceWidth()` to discover the width of the screen (in pixels) on which the app is running.

- Use `GetMaxDeviceHeight()` to discover the height of the screen (in pixels) on which the app is running.

- Use `GetDeviceWidth()` to discover the width of the window (in pixels) in which the app is running (desktop only).

- Use `GetDeviceHeight()` to discover the height of the window (in pixels) in which the app is running (desktop only).

- Use `MakeColor()` to construct an integer value holding red, green and blue colour information.

- Use `GetRed()`, `GetGreen()` and `GetBlue()` to extract the primary colour values from an integer holding colour information.

- Use `DrawLine()` to draw a line between two points. The line may morph from one colour to another along its length.

- Use `DrawBox()` to draw a rectangle between two points.

- The rectangle may be filled with the four colours defined filling each corner and merging towards the centre.

- The rectangle may be created in outline only with the four specified colours merging along the border.

- Use `DrawEllipse()` to draw ellipses and circles.

- Ellipses can be filled (merging two colours) or outlined.

Images

Introduction

Any additional visual or audio components 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 assigned 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 **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 x 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.
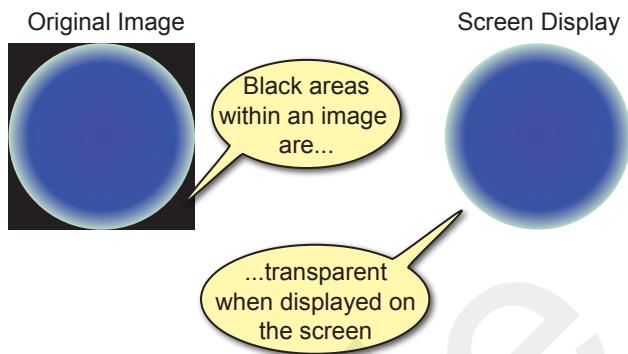
AGK BASIC recognises three image file formats. These are: BMP, PNG and JPG. BMP and PNG are lossless file formats and so should only be used for relatively small images; perhaps character figures and other visual components of a game. JPG is a lossy format and is ideal for use with photographs and larger 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-7.21).

Original Image

Black areas within an image are...

...transparent when displayed on the screen

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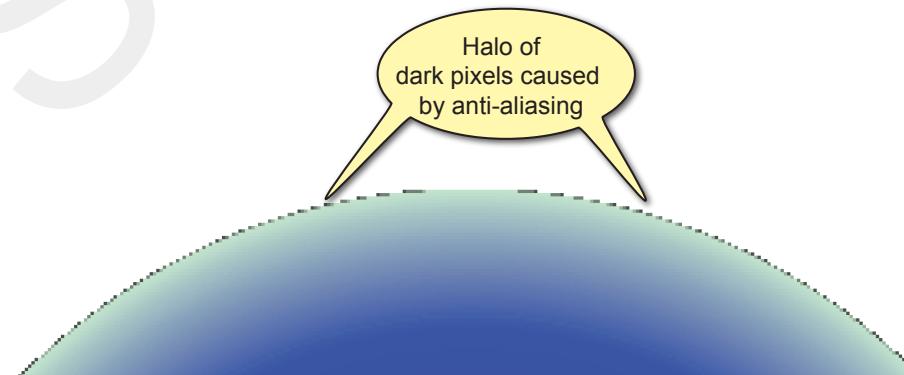
However, there are three things to be careful of when using this option:

- 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-7.22).

Halo of dark pixels caused by anti-aliasing

To avoid the halo problem, make sure anti-aliasing is switched off in your paint software 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.

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.

✎ JPG files cannot have an alpha channel.

FIG-7.23 shows an image, its alpha channel and how that image looks when displayed on screen.

FIG-7.23
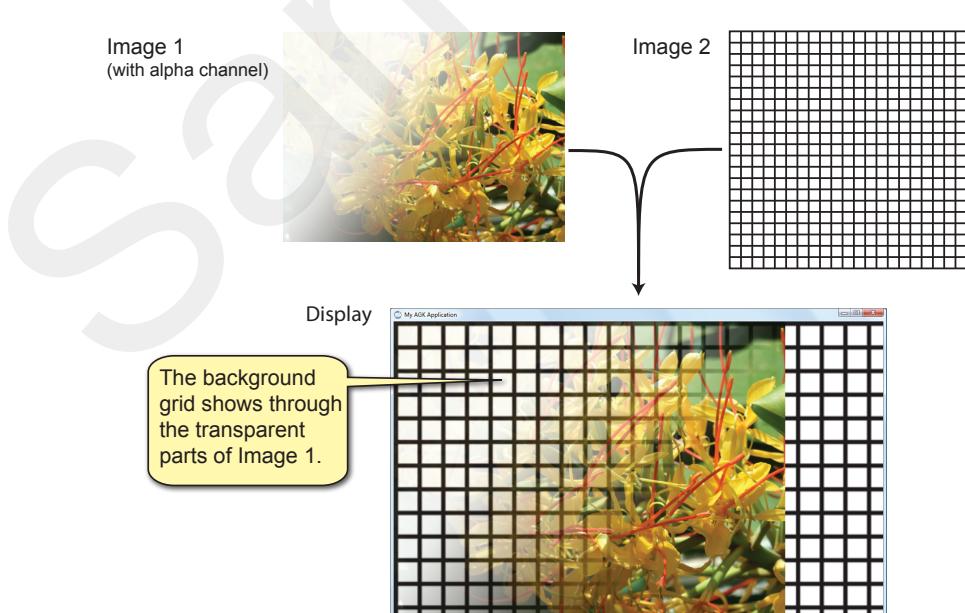
An Image with an Alpha Channel

Original   Alpha Channel   Displayed Image

The alpha channel determines transparency

The transparency is more obvious if we place a second image behind the original one (see FIG-7.24).

FIG-7.24

Alpha Channel Transparency

Image 1 (with alpha channel)   Image 2

Display

The background grid shows through the transparent parts of Image 1.

BMP and PNG files both allow alpha channel information to be stored (though in slightly different ways).

Images in AGK

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, within our program, we need to issue a command to load each image into the game itself. This is done using the `LoadImage()` function. There are two formats of this statement (see FIG-7.25).

FIG-7.25

LoadImage()

Version 1

LoadImage (id , file [, flag])

Version 2

integer LoadImage (file [, flag])

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.

file   is a string giving the name of the file containing the image. The file must be in the *media* folder for this project.

flag   is an integer (0 or 1) which is used to determine how transparency is handled when the image is displayed.

If *flag* 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.png*", then we would load the image using the statement

```
LoadImage(1,"Ball.png",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.png",1)
```

This time the program decides on the ID to be assigned, but IDs are assigned in ascending order starting at 100001, so, as long as this is the first image to be loaded it will be assigned an ID of 100001.

Using the second version guarantees that we will not accidentally attempt to assign the same ID to two different images (which would, in any case, produce an error).

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.

- Use `LoadImage()` to load an image and assign it a unique ID number.

Sprites

Introduction

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**.

The term *sprite* is used for a component containing a two-dimensional bitmap single frame or multi-frame image which can be positioned, sized, rotated and moved independently of other elements on the screen. When using a multi-frame image, cartoon-like animation can be achieved.
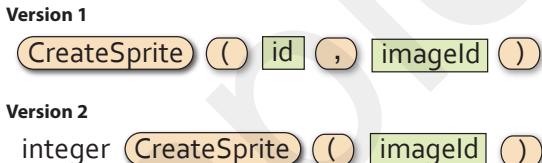
Using Sprites

CreateSprite()

To create a sprite we need to specify the image to be displayed by the sprite. This is done using the `CreateSprite()` statement (see FIG-7.26).

FIG-7.26

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.

Like the two versions of `LoadImage()`, the two options in the `CreateSprite()` statement allow us to choose between deciding on the ID number ourself (version 1) or letting the program decide for us (version 2 - assigned values start at 100001).

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.png",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-7.27).

FIG-7.27

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.

```
// Project: FirstSprite
// Created: 2015-01-16

//*** Set window title and size ***
SetWindowTitle("First Sprite")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Load image ***
LoadImage(1,"Ball.png")

//*** Create sprite ***
CreateSprite(1,1)
do
    Sync()
loop
```

Notice that the sprite is created outside the `do...loop` structure, and unlike `Print()` and `Draw...()` statements, there is no requirement to recreate the sprite each time the screen is refreshed.

Once a sprite has been created, AGK handles its display, making sure it is visible continually without any requirement from us to code for its reappearance between each call to `Sync()`.

> **Activity 7.7**
>
> 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 *AGK2/Resources/Ch07* folder and copy the file *Ball.png* to the project's *media* folder.
>
> Change the contents of *main.agc* to match that given in FIG-7.27.
>
> Run program. What is strange about the image?

As we can see from running *FirstSprite*, AGK has a problem with sizing the image (see FIG-7.28).

FIG-7.28

Sprite Size Problem

Since we are working with a percentage-based screen layout, AGK has no idea exactly how large to make the sprite. It handles this by assuming that the physical size of the image represents the percentage required. The ball image is 128 pixels wide by 128 pixels high, so AGK assumes you want the image to take up 128% of the width 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-7.29.

where:

id	is the integer value previously assigned as the ID of the sprite to be resized.
w	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.
h	is a real value giving the height required (percentage or virtual pixels).

So, if we wanted the ball sprite to occupy only 50% of the screen's width and height, we would use the line:

```
SetSpriteSize(1,50,50)
```
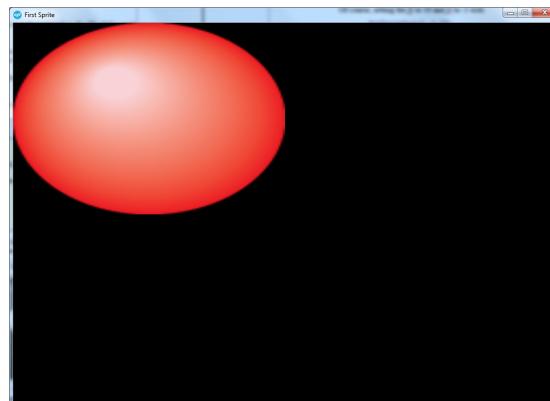
> **Activity 7.8**
>
> Modify *FirstSprite* by adding the `SetSpriteSize()` statement given above.
>
> Run the program and see how this changes the image displayed. What shape is the ball?

As you can see from Activity 7.8, making the sprite 50% in both directions causes the circular ball to become rugby-ball shaped (see FIG-7.30).

The reason for this is simple enough to work out when we remember that our window is 1024 pixels wide and 720 pixels high. Since the ball is 50% of the width and 50% of the height, that means that it is 512 pixels wide but only 360 pixels high!

Rather than work out the correct percentage for the sprite in order to make it 512 pixels high and hence return to a round-shaped ball, `SetSpriteSize()` allows us to set the actual size of one dimension and use the value -1 for the other. When we choose this option, AGK works out the second dimension automatically to ensure that the sprite retains its original width-to-height ratio. For example, if we set the *x* parameter to 50 and *y* to -1 using the line

```
SetSpriteSize(1,50,-1)
```

the sprite will return to its round shape.
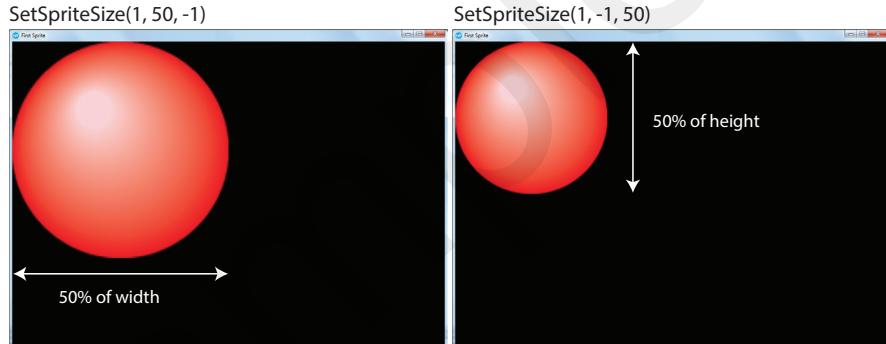
Of course, setting the *y* to 50 and *x* to -1 with

```
SetSpriteSize(1,-1,50)
```

will still result in a round ball, but this second statement will produce a ball that is smaller since 50% of the app window's height is much less than 50% of its width (see FIG-7.31).

FIG-7.31

How Sprite Size Changes with Screen Size

SetSpriteSize(1, 50, -1)   SetSpriteSize(1, -1, 50)

50% of width   50% of height

> **Activity 7.9**
>
> 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.

GetSpriteWidth()

We can discover the width of a sprite using the `GetSpriteWidth()` function (see FIG-7.32).

FIG-7.32

GetSpriteWidth()

float GetSpriteWidth (id)

where

id   is an integer value giving the ID of the sprite whose width is to be retrieved.

The value returned will be given in the units of measurement used by the program (percentage or virtual pixels).

GetSpriteHeight()

We can discover the height of a sprite using the `GetSpriteHeight()` function (see FIG-7.33).

$$\text{float} \quad \boxed{\text{GetSpriteHeight}} \; (\;) \; \boxed{\text{id}} \; (\;)$$

where

id	is an integer value giving the ID of the sprite whose width is to be retrieved.

The value returned will be given in the units of measurement used by the program (percentage or virtual pixels).

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-7.34.

$$\boxed{\text{SetSpritePosition}} \; (\;) \; \boxed{\text{id}} \; (,) \; \boxed{\text{x}} \; (,) \; \boxed{\text{y}} \; (\;)$$

where:

id	is the integer value previously assigned as the ID of the sprite to be moved.
x, y	are real values giving the new coordinates of the sprite (percentage or virtual pixels). These coordinates refer to the position of the top-left corner of the sprite.

Activity 7.10

In *FirstSprite*, modify the code to reduce the size of the ball to 10% of the app height. Create a two second delay then move the ball sprite to the centre of the app window.

Test 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 7.11

Remove the `SetSpritePosition()` call from *FirstSprite* and replace it with the following code:

```
for p = 1 to 100
        SetSpritePosition(1,p,p)
        Sync()
next p
```

Test the new version of the project.

GetSpriteX() and GetSpriteY()

We can discover the coordinates of a sprite's top left corner using the functions **GetSpriteX()** and **GetSpriteY()** which return the *x* and *y* coordinates respectively. The formats of the two statements are shown in FIG-7.35.

float (GetSpriteX) ( ) [id] ( )
float (GetSpriteY) ( ) [id] ( )

where

id   is an integer value giving the ID of an existing sprite.

The value returned will use the coordinate system setup for the program (percentage or virtual pixels).

> **Activity 7.12**
>
> Modify *FirstSprite* so that the latest position of the ball sprite is displayed and updated as the ball moves across the screen.
>
> Test your project.
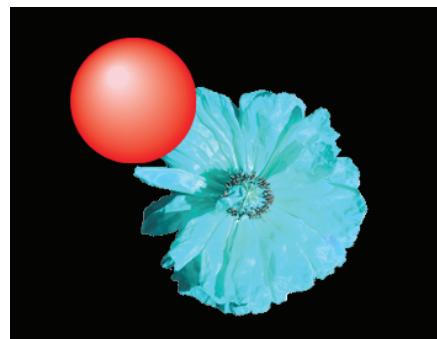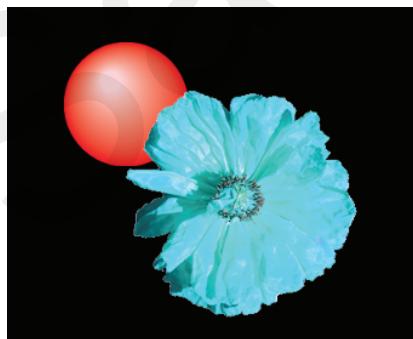
Sprite Depth

If two or more sprites overlap, the last one to be created will often appear "on top", obscuring some or all of the earlier sprite. For example, in FIG-7.36 we can see the results of overlapping a ball and poppy sprite with different creation sequences.

CreateSprite(1,1)   //Ball sprite   CreateSprite(2,2)   //Poppy sprite
CreateSprite(2,2)   //Poppy sprite   CreateSprite(1,1)   //Ball sprite

SetSortCreated()

However, AGK does not guarantee that sprites will behave in this way (with the last to be created appearing over the earlier one) unless a call is made to the **SetSortCreated()** function (see FIG-7.37).

(SetSortCreated) ( ) [flag] ( )

where

flag	is 0 or 1. When set to 1, sprites on the same depth layer (see below) are drawn in the order in which they were created. When set to 0 (the default value), the order in which sprites are drawn is undefined.

SetSpriteDepth()

Back in the days when animation cartoons were drawn by hand, this overlapping effect was achieved by drawing each image on a separate sheet of acetate, with the object on the top sheet obscuring objects on lower sheets.

AGK BASIC achieves the digital equivalent of these acetate sheets using sprite layers. The layer on which a sprite is placed can be set using the `SetSpriteDepth()` function. The layer specified when calling this function can range between 0 and 10,000, with 0 being the top layer and 10,000 the bottom layer.

By default, all sprites are placed on layer 10 (depth 10). If we want to ensure overlapping sprites are shown in a specific layered sequence, more efficient bytecode will result from setting each sprite to a different depth rather than leaving them at the same depth and making use of `SetSortCreated()`.

To set a sprite's layer, use `SetSpriteDepth()` (see FIG-7.38).

FIG-7.38

SetSpriteDepth()

where:

id	is the integer value previously assigned as the ID of the sprite.
depth	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.

The program in FIG-7.39 is an extension of your *FirstSprite* project and demonstrates one sprite passing "behind" another.

FIG-7.39

Demonstrating Sprite Depth

```
// Project: SpriteDepth
// Created: 2015-01-16

//*** Set window title and size ***
SetWindowTitle("Sprite Depth")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)
ClearScreen()

//*** Load images ***
ball_img = LoadImage("Ball.png")
poppy_img = LoadImage("Poppy.png")

//*** Create ball sprite ***
ball_spr = CreateSprite(ball_img)
SetSpriteSize(ball_spr,10,-1)

//*** Create poppy sprite on layer 9 ***
poppy_spr = CreateSprite(poppy_img)
SetSpriteSize(poppy_spr,20,-1)
```

FIG-7.39
(continued)

Demonstrating Sprite
Depth

```
SetSpritePosition(poppy_spr,40,40)
SetSpriteDepth(poppy_spr,9)

//*** Move ball sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(ball_spr,p,p)
    Sync()
next p

//*** Do nothing ***
do
    Sync()
loop
```

The layer on which a sprite is placed can be changed during the execution of a program. This allows us to have the sprite pass in front of a second sprite during one part of the program and behind the same sprite later.

> **Activity 7.13**
>
> Create a new project called *SpriteDepth* and code *main.agc* to match the code given in FIG-7.39.
>
> Copy the files *Ball.png* and *Poppy.png* from the AGK2/Resources/*Ch07* resources folder to the project's *media* folder.
>
> Test your program.
>
> Modify your program so that the ball repeats its movement from top left to bottom right continually.
>
> Test your project.

> **Activity 7.14**
>
> Modify *SpriteDepth* so that the ball passes over the poppy rather than beneath it. Test your project.

GetSpriteDepth()

There will sometimes be situations where we don't know which layer a sprite is on (perhaps its layer has been chosen at random); to determine the current depth of a sprite, use the `GetSpriteDepth()` statement (see FIG-7.40) which returns a sprite's depth setting.

FIG-7.40

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-7.41.

Version 1

CloneSprite ( ) id , idToCopy)

Version 2

integer CloneSprite ( ) 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.

The two options in the `CloneSprite()` statement allow us to choose between deciding on the ID number ourself (version 1) or letting the program decide for us (version 2).

Whatever characteristics have been set for the original sprite (size, transparency, depth, etc.) will be duplicated in the clone.

> **Activity 7.15**
>
> Modify *SpriteDepth*, 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?

SetSpriteVisible()

We can make a sprite invisible – and make it reappear – using the `SetSpriteVisible()` statement which has the format shown in FIG-7.42.

SetSpriteVisible ( ) id , visible)

where:

id	is the integer value previously assigned as the ID of the sprite.
visible	is an integer value (0 or 1) specifying that the sprite is to be hidden (0) or made visible (1).

> **Activity 7.16**
>
> Modify *SpriteDepth* so that the two poppy sprites are hidden after the ball has moved to the bottom of the screen for the first time.

There is an inefficiency in the solution's code, in that the new lines are repeatedly executed, when, in fact, they need only be executed once. A better (but longer) solution would be to end the program with the following code:

```
//*** Move ball sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(ball_spr,p,p)
    Sync()
next p

//*** Make poppies invisible ***
SetSpriteVisible(poppy_spr,0)
SetSpriteVisible(poppy2_spr,0)

do
    //*** Move ball sprite across the screen ***
    for p = 1 to 100
        SetSpritePosition(ball_spr,p,p)
        Sync()
    next p
loop
```

Now the first traversal of the ball and the hiding of the poppies are dealt with before entering the **do...loop** and, as a result, are only executed once.

GetSpriteVisible()

To discover if a sprite is currently visible, we can use the **GetSpriteVisible()** statement which has the format shown in FIG-7.43.

FIG-7.43

GetSpriteVisible()

where:

id   is the integer value previously assigned as the ID of the sprite.

The function returns 1 if the sprite is visible, 0 if it is not.

SetSpriteAngle() and SetSpriteAngleRad()

We can rotate a sprite by a specified angle using either **SetSpriteAngle()** – which accepts an angle given in degrees – or **SetSpriteAngleRad()** – which takes an angle given in radians. The format for these functions is shown in FIG-7.44.

FIG-7.44

SetSpriteAngle()
SetSpriteAngleRad()

where

id   is an integer value giving the ID of the sprite to be rotated.

deg   is a real value giving the angle (in degrees) through which the sprite is to be rotated.

| **rad** | is a real value giving the angle (in radians) through which the sprite is to be rotated. |

The angle given is an absolute value – not relative to the sprite's current rotation – and is measured from the 3 o'clock position in a clockwise direction.

For example, an arrow shaped sprite, position near the centre of the app window, and rotated by 45°, creates the display shown in FIG-7.45.

FIG-7.45

A Sprite Rotated by 45°

The program in FIG-7.46 rotates the arrow sprite shown above 1° at a time to create a revolving effect.

FIG-7.45

A Sprite Rotated by 45°

```
// Project: RotateSprite
// Created: 2015-02-26

//*** Set window size and title ***
SetWindowSize(1024,768,0)
SetWindowTitle("Rotate a Sprite")

//*** Clear the screen to grey ***
SetClearColor(200,200,200)
ClearScreen()

//*** Set up sprite ***
img = LoadImage("Arrow.png")
spr = CreateSprite(img)
SetSpriteSize(spr,20,-1)
SetSpritePosition(spr,40,50)

//*** Display continually rotating sprite ***
do
    for angle = 0 to 359
        SetSpriteAngle(spr,angle)
        Sync()
    next angle
loop
```

GetSpriteAngle() and GetSpriteAngleRad()

To discover the current angle of rotation of a sprite we can use the functions `GetSpriteAngle()` and `GetSpriteAngleRad()` (see FIG-7.46).

FIG-7.46

GetSpriteAngle()
GetSpriteAngleRad()

float (GetSpriteAngle) ( ) [id] ( )
float (GetSpriteAngleRad) ( ) [id] ( )

where

id   is an integer value giving the ID of the sprite whose rotation is to be determined.

The value returned by the functions is the angle through which the sprite is currently rotated in either degrees or radians depending on which function is used.

SetSpriteOffset()
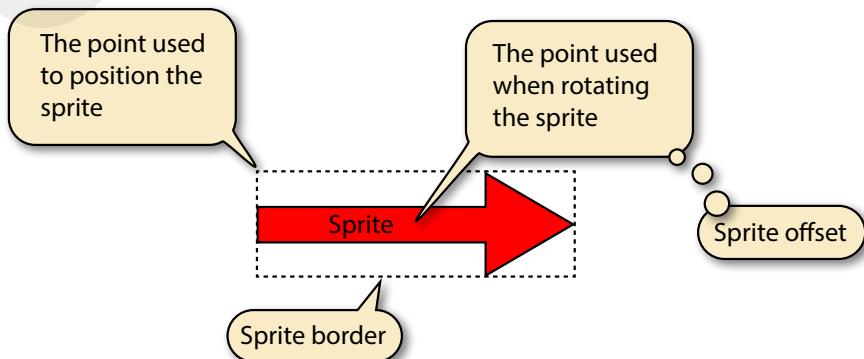
By default, when a sprite is positioned, it is the top-left corner of that sprite that is located at the given position, but when a sprite is rotated, it rotates about the centre of the sprite. The point about which the sprite rotates is known as the **sprite offset.** (see FIG-7.47).

FIG-7.47

Points Used for Positioning and Rotation

The sprite's point of rotation can be repositioned using the `SetSpriteOffset()`

function (see FIG-7.48).

$$\boxed{\text{SetSpriteOffset}} \; \boxed{(} \; \boxed{\text{id}} \; \boxed{,} \; \boxed{\text{x}} \; \boxed{,} \; \boxed{\text{y}} \; \boxed{)}$$

where

id	is an integer value giving the ID of the sprite whose offset is to be modified.
x, y	are real values giving the position of the new offset. These values are measured from the top-left corner of the sprite which is taken as point (0,0) when specifying values for *x* and *y*.
	The offset point may be outside the bounds of the sprite.
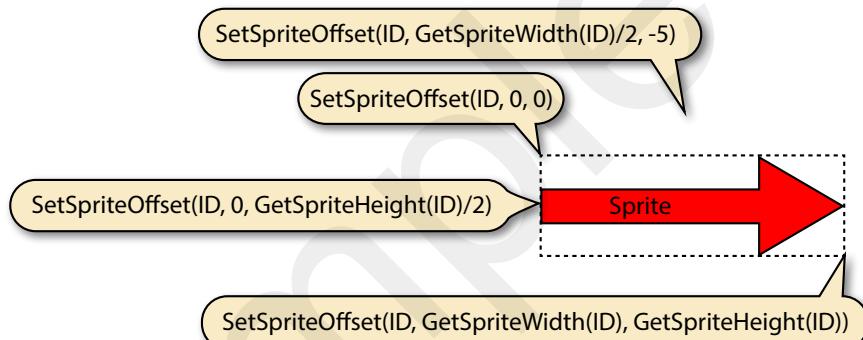
Various possible offset points are shown in FIG-7.49.

Activity 7.19

Modify *RotateSprite* so that sprite rotates about the middle of its left edge.

GetSpriteOffsetX() and GetSpriteOffsetY()

To discover a sprite's current offset values (measured from the top-left of the sprite), we can use the commands **GetSpriteOffsetX()** and **GetSpriteOffsetY()** (see FIG-7.50).

$$\text{float} \; \boxed{\text{GetSpriteOffsetX}} \; \boxed{(} \; \boxed{\text{id}} \; \boxed{)}$$
$$\text{float} \; \boxed{\text{GetSpriteOffsetY}} \; \boxed{(} \; \boxed{\text{id}} \; \boxed{)}$$

where

id	is an integer value giving the ID of the sprite.

GetSpriteXByOffset() and GetSpriteYByOffset()

The functions **GetSpriteXByOffset()** and **GetSpriteYByOffset()** return the screen coordinates of a sprite's offset point. These functions have the format shown in FIG-7.51.

FIG-7.51

GetSpriteXByOffset()
GetSpriteYByOffset()

float (GetSpriteXByOffset) ( ) id ( )

float (GetSpriteYByOffset) ( ) id ( )

where

id   is an integer value giving the ID of the sprite.

> **Activity 7.20**
>
> Modify *RotateSprite* so that `Print()` statements display the sprite's offset values and the screen position of the offset point.

SetSpritePositionByOffset()

When we use `SetSpritePosition()` to place a sprite on the screen, the coordinates we specify are those for the top-left corner of the sprite. However, by using `SetSpritePositionByOffset()` we can position the sprite according to its offset position. The format for this statement is given in FIG-7.52.

FIG-7.52

SetSpritePositionBy
↳Offset()

(SetSpritePositionByOffset) ( ) id , x , y ( )

id   is the integer value previously assigned as the ID of the sprite to be moved.

x, y   are real values giving the new coordinates of the sprite (percentage or virtual pixels). These coordinates refer to the position of the offset of the sprite.

For example, assuming a sprite (ID = 1) has its default offset setting (at the sprite's centre), then using the line
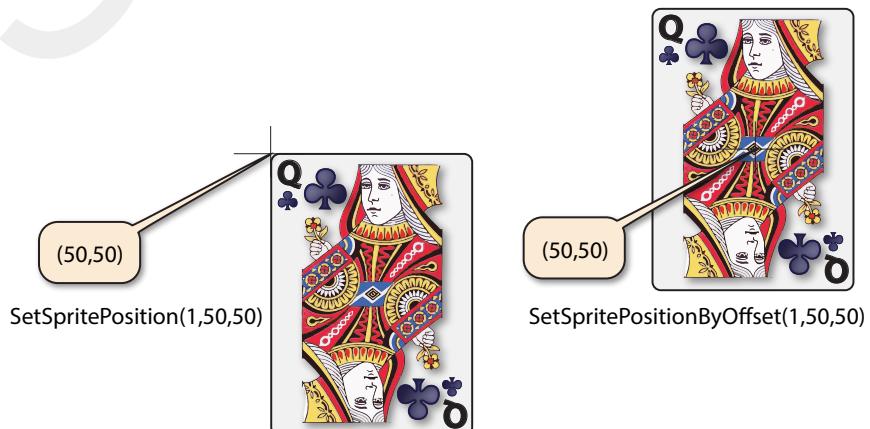
```
SetSpritePositionByOffset(1, 50, 50)
```

would place the sprite's centre at position (50,50).

FIG-7.53 shows the difference between using `SetSpritePosition()` and `SetSpritePositionByOffset()` to position a sprite.

FIG-7.53

Using
SetSpritePositionByOffset()

SetSpritePosition(1,50,50)   SetSpritePositionByOffset(1,50,50)

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 our game. Sprites are deleted using the `DeleteSprite()` statement whose format is shown in FIG-7.54.

FIG-7.54

DeleteSprite()

$$\boxed{\text{DeleteSprite}} \; \boxed{(} \; \boxed{\text{id}} \; \boxed{)}$$

where:

   id   is an integer value giving the ID of the sprite to be deleted.

DeleteAllSprites()

If a program contains several sprites, they can all be deleted, using the `DeleteAllSprites()` statement (see FIG-7.55).

FIG-7.55

DeleteAllSprites()

$$\boxed{\text{DeleteAllSprites}} \; \boxed{(} \; \boxed{)}$$

DeleteImage()

When an image is no longer required by any sprite, that image can be deleted, thereby freeing up further resources. To delete an image we use the `DeleteImage()` statement (see FIG-7.56).

FIG-7.56

DeleteImage()

$$\boxed{\text{DeleteImage}} \; \boxed{(} \; \boxed{\text{id}} \; \boxed{)}$$

where:

   id   is an integer value giving the ID of the image to be deleted.

DeleteAllImages()

Rather than delete images individually, you can delete every loaded image using the `DeleteAllImages()` statement (see FIG-7.57).

FIG-7.57

DeleteAllImages()

$$\boxed{\text{DeleteAllImages}} \; \boxed{(} \; \boxed{)}$$

Of course, you should only call this statement when every image in the program is no longer being used by other program elements such as a sprite.

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.

Summary

- To display an image on the screen it must first be loaded into a sprite.

- Use `CreateSprite()` to create a sprite from a previously loaded image.

- Using the default setup, screen distances are given in percentage terms and

sprites use the pixel dimensions of the image it contains as a percentage value when determining the size of the image.

- Use `SetSpriteSize()` to set the size of a sprite.

- When sizing a sprite, use a value of -1 for the width (or height) in order to allow AGK to maintain the correct width-to-height ratio for the image displayed in the sprite.

- Use `GetSpriteWidth()` and `GetSpriteHeight()` to find the current dimensions of a sprite.

- Use `SetSpritePosition()` to position a sprite.

- The coordinates given when positioning a sprite are for the top-left corner of the sprite.

- 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 be drawn in front of a sprite placed on a lower layer.

- When sprites are placed on the same layer, the order in which they are drawn is, by default, undefined.

- Use `SetSortCreated()` to ensure sprites on the same layer are drawn in the order they are created (latest sprite in front of earlier sprites).

- Use `SetSpriteDepth()` to set the layer on which a sprite is to be drawn.

- Use `GetSpriteDepth()` to discover the layer on which a sprite has been drawn.

- Use `CloneSprite()` to create an exact copy of an existing sprite.

- A cloned sprite will initially be an exact copy of the original in terms of sprite size, draw layer, visibility, etc.

- Use `SetSpriteVisible()` to make a sprite invisible/visible.

- Use `GetSpriteVisible()` to determine the visibility of a specified sprite.

- Use `SetSpriteAngle()` or `SetSpriteAngleRad()` to set a sprite's angle of rotation.

- Use `GetSpriteAngle()` or `GetSpriteAngleRad()` to get a sprite's angle of rotation.

- Use `SetSpriteOffset()` to modify the point of rotation of a sprite.

- Use `GetSpriteOffsetX()` and `GetSpriteOffsetY()` to discover a sprite's offset position measured from the top-left of the sprite.

- Use `GetSpriteXByOffset()` and `GetSpriteYByOffset()` to discover a sprite's offset screen coordinates.

- Use `SetSpritePositionByOffset()` to position a sprite according to its current offset position.

- Use `DeleteSprite()` to delete a specific sprite.

- Use `DeleteAllSprites()` to delete every sprite.

- If the sprite displaying an image has been deleted, the image shown on that sprite (assuming it is not displayed on other sprites) can be deleted.

- Use `DeleteImage()` to delete a specified, previously loaded image.

- Use `DeleteAllImages()` to delete all images previously loaded.

- Deleting sprites and images frees up memory.

Detecting User Interaction

Introduction

Most video games react to the user clicking a mouse or touching a pressure-sensitive screen. AGK BASIC uses three main commands to detect a mouse or screen press. Using these statements we can allow the user to interact with the program. Typically, we will detect a press or release to activate a button or affect the behaviour of a sprite.

Pointer Statements

GetPointerPressed()

We can detect the moment a mouse left button press (or a screen press) happens using the `GetPointerPressed()` function which has the format shown in FIG-7.58.

FIG-7.58

GetPointerPressed()

integer (GetPointerPressed) (( ) ( ))

The statement returns 1 if a press has just occurred. Before and after that instant, zero is returned. Note that this means that the function will return zero if called after the initial press, even though the button/screen continues to be pressed.

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-7.59.

FIG-7.59

GetPointerReleased()

integer (GetPointerReleased) (( ) ( ))

If a release has not occurred at that instant, zero is returned.

The code in FIG-7.60 demonstrates the use of the `GetPointerPressed()` and `GetPointerReleased()` statements, displaying a message each time a press or release is detected.

FIG-7.60

Using Pointer Statements

```
// Project: UserInteraction
// Created: 2015-01-17

//*** Set window title and size ***
SetWindowTitle("User Interaction")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)
ClearScreen()

do
   //*** Check for press ***
    if GetPointerPressed()=1
        Print("Pressed")
    endif
   //*** Check for release ***
    if GetPointerReleased()=1
        Print("Released")
    endif
    Sync()
loop
```

GetPointerState()

Sometimes, we are not interested the moment a press or release occurs but are more concerned with the general state of a button – whether it is currently being held down or not. The **GetPointerState()** function returns 1 while the left mouse button or finger is being pressed down and returns 0 when the button/finger is not pressed.

The **GetPointerState()** command has the format shown in FIG-7.61.

integer (GetPointerState) (( ) ( ))

Note this is different from the first two statements which only return 1 for a single instant as the mouse/finger is pressed/lifted.

GetPointerX() and GetPointerY()

We can find out the current position on the screen of the mouse pointer or discover the last position where the screen has been touched using **GetPointerX()** (which returns the *x*-coordinate) and **GetPointerY()** (which returns the *y*-coordinate).

The formats for these two statements are shown in FIG-7.62.

float (GetPointerX) (( ) ( ))

float (GetPointerY) (( ) ( ))

The program in FIG-7.63 displays the coordinates of the pointer as it is moved about the screen.

FIG-7.63

Displaying the Pointer's
Coordinates

```
// Project: PointerPosition
// Created: 2015-01-17


//*** Set window title and size ***
SetWindowTitle("Pointer Position")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen to light grey ***
SetClearColor(200,200,200)
ClearScreen()

do
    //*** Get pointer's coordinates ***
    x# = GetPointerX()
    y# = GetPointerY()

    //*** Display pointer coordinates ***
    PrintC("(")
    PrintC(x#)
    PrintC(',')
    PrintC(y#)
    Print(")")
    Sync()
loop
```

Activity 7.23

Create a new project called *PointerPosition* and code *main.agc* to match the code given in FIG-7.63.

Run the program. Is it possible to position the pointer at points (0,0) and (100,100)?

Try running the app on your Android tablet using *AGK Player 2*. What range of points can be achieved?

✎
The results achieved by the second part of Activity 7.19 depend entirely on the aspect ratio of your tablet.

Rather than have a continually updating coordinate display, we could read the coordinates of the pointer only at the moment the mouse button (or screen) is pressed by making use of the code:

```
if GetPointerPressed() = 1
    x# = GetPointerX()
    y# = GetPointerY()
endif
```

Activity 7.24

Modify *PointerPosition* so that the pointer coordinates are only updated when the mouse button is pressed.

Test your program.

The Screen Pointer and Sprites

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-7.64.

$$\text{integer} \quad \boxed{GetSpriteHit} \; (\quad) \quad \boxed{x} \quad , \quad \boxed{y} \quad (\quad)$$

where:

      x, y   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.

The program in FIG-7.65 displays two sprites: *ball* and *poppy*. When the mouse pointer moves over a sprite (or the screen is pressed over a sprite), that sprite becomes invisible.

```
// Project: SpriteOver
// Created: 2015-01-17

//*** Set window title and size ***
SetWindowTitle("Sprite Over")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Set screen background to grey ***
SetClearColor(120,120,120)
ClearScreen()

//*** Load images required ***
ball_img = LoadImage("Ball.png")
poppy_img = LoadImage("Poppy.png")

//*** Create, size and position ball sprite ***
ball_spr = CreateSprite(ball_img)
SetSpriteSize(ball_spr, 10,-1)
SetSpritePosition(ball_spr, 20, 33)

//*** Create, size and position poppy sprite ***
poppy_spr = CreateSprite(poppy_img)
SetSpriteSize(poppy_spr, 15,-1)
SetSpritePosition(poppy_spr, 60, 30)

do
   //*** Get ID of any sprite under pointer ***
   spr_hit = GetSpriteHit(GetPointerX(),GetPointerY())

   //*** If a sprite is under pointer, hide it ***
   if spr_hit <> 0
      SetSpriteVisible(spr_hit,0)
   endif
   Sync()
loop
```

Activity 7.25

Create a new project called *SpriteOver* and code *main.agc* to match the code given in FIG-7.65. Run the program and move the pointer over each sprite in turn. Do the sprites disappear when the pointer moves over them?

Using `GetPointerPresssed()`, modify the program so that a sprite only disappears when the mouse button is clicked over it.

Test your program.

We might be forgiven for assuming that when a sprite is invisible it can no longer be hit. That is to say, `GetSpriteHit()` would not detect the fact that the pointer was over that invisible sprite. But, in fact, this is not the case. Invisible sprites are detected in just the same way as visible ones.

We can get a sprite in our last program to flip between being visible and invisible using the expression

```
SetSpriteVisible(spr_hit.1-GetSpriteVisible(spr_hit))
```

Activity 7.26

Modify *SpriteOver* so that the sprites switch between being visible and invisible each time they are clicked.

Test your program.

Summary

- 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.

- Use `GetPointerPressed()` to check if a mouse button or screen press has just taken place.

- Use `GetPointerReleased()` to check if a mouse button has just been released, or a finger lifted from a screen.

- Use `GetPointerState()` to check if a mouse button is being held down, or a finger remains pressed on the screen.

- Use `GetPointerX()` and `GetPointerY()` to determine the current location of the mouse pointer on the screen or the last point touched on a screen.

- Use `GetSpriteHit()` to determine if a specified point on the screen is over an existing sprite.

- The value returned by `GetSpriteHit()` is unaffected by the visibility of a sprite.

Text Resources

Introduction

We've already seen how to display information on the screen using the `Print()` statement. But the output produced by the `Print()` statement has two main disadvantages:

- it disappears after subsequent calls to `Sync()` and so the `Print()` statement must be executed repeatedly to maintain the text on the screen.

- there is no control over where on the screen the text will appear.

Both these limitations mean that the `Print()` statement is of little use in most apps.

Luckily, AGK offers a second and more controlled way of creating textual output – **text resources**. Just like image and sprite resources, text resources must be created and are assigned a unique ID.

Using a text resource, we can position text anywhere on the screen and, as with sprites, AGK automatically ensures that the text remains visible after calls to `Sync()`.

A few of the many statements available for manipulating text resources are described here.
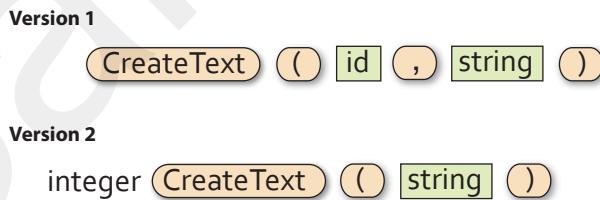
Text Statements

CreateText()

The `CreateText()` statement allows us to create a new text resource. The statement has the format shown in FIG-7.66.

FIG-7.66

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-7.67

SetTextColor()

By default, text displayed by a text resource is white, but we can select the color and transparency of a specific text resource using the `SetTextColor()` statement (see FIG-7.67).

SetTextColor (id , red , green , blue , opac)

where:

id   is an integer value specifying the ID of the text resource whose colour is to be set.

The default colour for a text resource is white.

red   is an integer value specifying the intensity of the red component of the colour. Range 0 to 255.

green   is an integer value specifying the intensity of the green component of the colour. Range 0 to 255.

blue   is an integer value specifying the intensity of the blue component of the colour. Range 0 to 255.

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

For example, if we have already created a text resource with an ID of 1, then we can display that text in opaque black using the line:

```
SetTextColor(1,0,0,0,255)
```

SetTextPosition()

FIG-7.68

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-7.68).

SetTextPosition (id , x , y)

where:

id   is the integer value previously assigned as the ID of the text to be moved.

x,y   are a real values giving the new coordinates for the specified text resource. This will be in virtual pixels or percentage depending on the coordinate system defined when the app window was created.
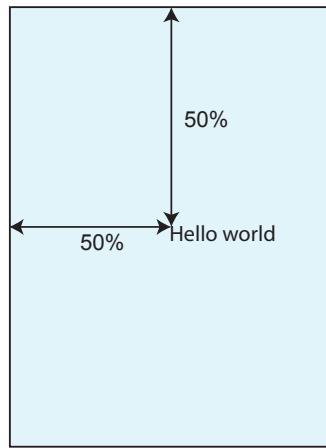
A text resource whose ID value is 1 could be placed at the centre of the app window using the statement:

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

The position (50,50) refers to the top left part of the text (see FIG-7.69).

FIG-7.69

Positioning a Text
Resource
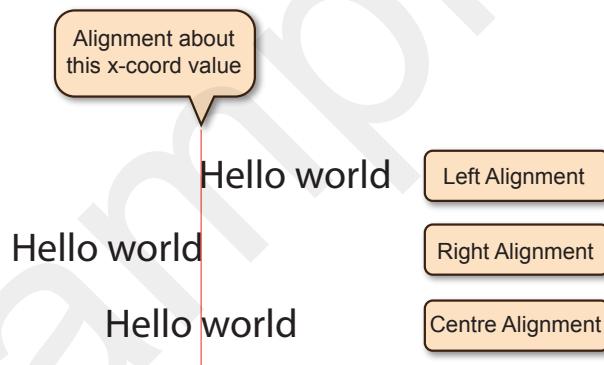
SetTextAlignment()

By default, the text position specifies where the top left corner of the text is to be positioned. This is known as left-aligned text. But it is possible to modify this so that the position is used to place the top-right corner of the text (right-aligned text). Finally, the text can be centre-aligned using the position given to place the top-centre of the text. FIG-7.70 shows the effect of each alignment option.

FIG-7.70

Alignment Options

FIG-7.71

SetTextAlignment()

To change from the default left-aligned text, use the `SetTextAlignment()` statement (see FIG-7.71).

where:

id	is an integer value giving the ID of the text resource.
align	is an integer value (0, 1 or 2) which gives the alignment to be used (0: left-alignment, 1: centre-alignment, 2: right-alignment). The default is zero.

Activity 7.27

Start a new project called *TextAlignment* and display three instances of the words "*Hello world*" each with a different alignment. Set the x coordinate to 50 for all three text resources.

SetTextSize()

The size of the text can be adjusted using the `SetTextSize()` statement (see FIG-7.72).

SetTextSize ( ) id , size ( )

where:

id	is the integer value previously assigned as the ID of the text to be resized.
size	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.

The characters displayed in a text resource are created using an image of the character set. As the text is made larger, so the low resolution of the image used becomes more obvious creating a slightly blurred look to the text. In a later chapter we'll see how to use our own font images.

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-7.73).

SetTextString ( ) id , string ( )

where:

id	is the integer value giving the text's ID.
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-7.74).

SetTextVisible ( ) id , visible ( )

where:

id	is the integer value previously assigned as the ID of the text resource to be operated on.

| | **visible** | is an integer value (0 or 1) used to hide or display the text. (0 - hidden ; 1 - visible) |

GetTextVisible()

To discover if a text resource is currently visible, we can use the `GetTextVisible()` statement which has the format shown in FIG-7.75.

$$\text{integer} \quad \boxed{\text{GetTextVisible}} \; (\;) \; \boxed{\text{id}} \; (\;)$$

where:

| | **id** | is the integer value previously assigned as the ID of the text. |

The function returns 1 if the text is visible, 0 if it is not.

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-7.76).

$$\boxed{\text{DeleteText}} \; (\;) \; \boxed{\text{id}} \; (\;)$$

where:

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

DeleteAllText()

If your program contains several text resources and you wish to remove all of them, use `DeleteAllText()` (see FIG-7.77).

$$\boxed{\text{DeleteAllText}} \; (\;) \; (\;)$$

Using a Text Resource

The program in FIG-7.78 is an extension of the previous *SpriteOver* project with a text object above each sprite indicating whether that sprite is visible or not.

```
// Project: SpriteOver
// Created: 2015-01-17

//*** Set window title and size ***
SetWindowTitle("Sprite Over")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Set screen background to grey ***
SetClearColor(120,120,120)
ClearScreen()
```

FIG-7.78
(continued)

Using a Text Resource

```
//*** Load images required ***
ball_img = LoadImage("Ball.png")
poppy_img = LoadImage("Poppy.png")

//*** Create, size and position ball sprite ***
ball_spr = CreateSprite(ball_img)
SetSpriteSize(ball_spr, 10,-1)
SetSpritePosition(ball_spr, 20, 33)
//*** Create, size and position poppy sprite ***
poppy_spr = CreateSprite(poppy_img)
SetSpriteSize(poppy_spr, 15,-1)
SetSpritePosition(poppy_spr, 60, 30)

//*** Create text over ball ***
ball_txt = CreateText("Visible")
SetTextPosition(ball_txt, 20, 20)

//*** Create text over poppy ***
poppy_txt = CreateText("Visible")
SetTextPosition(poppy_txt, 60, 20)

do
   //*** If pointer pressed ***
   if GetPointerPressed() = 1
      //*** Get ID of any sprite under pointer ***
      spr_hit = GetSpriteHit(GetPointerX(),GetPointerY())
      //*** If a sprite is under pointer ***
      if spr_hit <> 0
         //*** toggle its visibility ***
         SetSpriteVisible(spr_hit,1-GetSpriteVisible(spr_hit))
         //*** If it's the ball ***
         if spr_hit = ball_spr
            //*** Set ball text ***
            if GetSpriteVisible(spr_hit) = 1
               SetTextString(ball_txt,"Visible")
            else
               SetTextString(ball_txt, "Invisible")
            endif
         else // it's the poppy
            //*** Set the poppy text ***
            if GetSpriteVisible(spr_hit) = 1
               SetTextString(poppy_txt,"Visible")
            else
               SetTextString(poppy_txt, "Invisible")
            endif
         endif
      endif
   endif
   Sync()
loop
```

Activity 7.28

Modify *SpriteOver* to match the code given in FIG-7.78.

Test the program, checking that each text correctly reflects the state of the sprite.

Summary

■ 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.

■ Use `CreateText()` to create a text resource.

■ Use `SetTextColor()` to set the colour of a specified text resource.

■ Use `SetTextPosition()` to position a specified text resource on the screen.

■ By default, the position specified in `SetTextPosition()` is applied to the top-left corner of the text.

■ Use `SetTextAlignment()` to change the default left alignment to right or centre alignment.

■ By default, text is white.

■ Use `SetTextSize()` to set the size of a specified text resource.

■ Use `SetTextString()` to change the text held in a text resource.

■ Use `SetTextVisible()` to set a specified text resource invisible/visible.

■ Use `GetTextVisible()` to determine if a specified text resource is visible.

■ Use `DeleteText()` to delete a specified text resource.

■ Use `DeleteAllText()` to delete a text resources.

Later

There are many other resource and input commands which are not covered here. These will be explained in later chapters.

Solutions

Activity 7.1

Most tablets and phones assume a portrait mode setup, so the width value will be less than the height.

The values from a mobile device do not change when it is turned to a different orientation.

Activity 7.2

Modified code for *ScreenSize*:

```
// Project: ScreenSize
// Created: 2015-01-23

//*** Window title and size ***
SetWindowTitle("Screen Size")
SetWindowSize(1024,500,0)
SetDisplayAspect(1024/500.0)

//*** Clear the screen ***
ClearScreen()

//*** Get screen dimensions ***
screenwidth = GetMaxDeviceWidth()
screenheight = GetMaxDeviceHeight()

//*** Get window dimensions ***
windowwidth = GetDeviceWidth()
windowheight = GetDeviceHeight()

do
    //*** Display actual dimensions ***
    PrintC("Screen width: ")
    PrintC(screenwidth)
    PrintC(" pixels       Screen height: ")
    PrintC(screenheight)
    Print(" pixels")
    //*** Display window dimensions ***
    PrintC("Window width: ")
    PrintC(windowwidth)
    PrintC(" pixels       Window height: ")
    PrintC(windowheight)
    Print(" pixels")
    Sync()
loop
```

The screen and window sizes are identical when the app is run on a portable device.

Activity 7.3

Code for *PercentPixel*:

```
// Project: PercentPixel
// Created: 2015-01-24

//*** Window title and size ***
SetWindowTitle("Percent/Pixel")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Calculate pixels per percent ***
one_percent_x# = GetDeviceWidth()/100.0
one_percent_y# = GetDeviceHeight()/100.0

//*** Calulate percent per pixel ***
xpixel# = 100.0/GetDeviceWidth()
ypixel# = 100.0/GetDeviceHeight()

do
    //*** Calculate 1% in pixels ***
    PrintC("1% in the x direction is ")
    PrintC(one_percent_x#)
    Print(" pixels")
    PrintC("1% in the y direction is ")
```

```
    PrintC(one_percent_y#)
    Print(" pixels")
    //*** Calculate 1 pixel as % ***
    PrintC("1 pixel represents ")
    PrintC(xpixel#)
    Print("% in the x direction")
    PrintC("1 pixel represents ")
    PrintC(ypixel#)
    Print("% in the y direction")
    Sync()
loop
```

Activity 7.4

Modified code for *TestDrawLine*:

```
// Project: TestDrawLine
// Created: 2015-01-23

//*** Window title and size ***
SetWindowTitle("Test DrawLine")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Create colour values ***
red = MakeColor(255,0,0)
yellow = MakeColor(255,255,0)

do
    //*** Draw a line from top-left to bottom-right
    ⤷***
    DrawLine(0,0,100,100,red, yellow)
    Sync()
loop
```

Activity 7.5

Modified code for *Rectangles*:

```
// Project: Rectangles
// Created: 2015-01-23

//*** Window title and size ***
SetWindowTitle("Rectangles")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

do
    //*** Create random colour values ***
    col1 = MakeColor(Random(0,255),Random(0,255),
    ⤷Random(0,255))
    col2 = MakeColor(Random(0,255),Random(0,255),
    ⤷Random(0,255))
    col3 = MakeColor(Random(0,255),Random(0,255),
    ⤷Random(0,255))
    col4 = MakeColor(Random(0,255),Random(0,255),
    ⤷Random(0,255))
    //*** Draw rectangle ***
    DrawBox(10,10,90,90,col1,col2,col3,col4,1)
    Sync()
    Sleep(500)
loop
```

Activity 7.6

Modified code for *Ecllipses*:

```
// Project: Ellipses
// Created: 2015-01-14

//*** Window title and size ***
SetWindowTitle("Ellipses")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Calculate how many pixels in 20% x direction
xpixels# = 20* GetDeviceWidth()/100

//*** Calculate % in y direction for same number of
pixels ***
ypercentage# = xpixels#/GetDeviceHeight() *100

do
    //*** Draw circle ***
```

```
DrawEllipse(50,50,20,ypercentage#,
 ⤷MakeColor(Random(0,255), Random(0,255),
 ⤷Random(0,255)),MakeColor(Random(0,255),
 ⤷Random(0,255), Random(0,255)),1)
Sync()
Sleep(500)
loop
```

Activity 7.7

Although the image is only 128 x 128 pixels it appears much larger within the app window.

Activity 7.8

Modified code for *FirstSprite*:

```
// Project: FirstSprite
// Created: 2015-01-16

//*** Set window title and size ***
SetWindowTitle("First Sprite")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Load image ***
LoadImage(1, "Ball.png")

//*** Create sprite ***
CreateSprite(1,1)

//*** Resize sprite ***
SetSpriteSize(1,50,50)

do
    Sync()
loop
```

The sprite now occupies 50% of the width and height of the app window. Because the app window is not square, this means that the ball is not perfectly round.

Activity 7.9

The line

```
SetSpriteSize(1,50,50)
```

should first be changed to

```
SetSpriteSize(1,50,-1)
```

The ball will be round.

On the next run the line should now read

```
SetSpriteSize(1,-1,50)
```

which will make the round ball's size 50% of the app window's height. But, because the window is not as tall as it is wide, the ball will be smaller than before.

Activity 7.10

Modified code for *FirstSprite*:

```
// Project: FirstSprite
// Created: 2015-01-16

//*** Set window title and size ***
SetWindowTitle("First Sprite")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Load image ***
LoadImage(1, "Ball.png")
```

```
//*** Create sprite ***
CreateSprite(1,1)

//*** Resize sprite ***
SetSpriteSize(1,-1,10)

//*** Display the sprite ***
Sync()

//*** Wait two seconds ***
Sleep(2000)

//*** Move the sprite to (50,50) ***
SetSpritePosition(1,50,50)

do
    Sync()
loop
```

Activity 7.11

Modified code for *FirstSprite*:

```
// Project: FirstSprite
// Created: 2015-01-16

//*** Set window title and size ***
SetWindowTitle("First Sprite")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Load image ***
LoadImage(1, "Ball.png")

//*** Create sprite ***
CreateSprite(1,1)

//*** Resize sprite ***
SetSpriteSize(1,-1,10)

//*** Display the sprite ***
Sync()

//*** Wait two seconds ***
Sleep(2000)

//*** Move the sprite across screen ***
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
next p

do
    Sync()
loop
```

Activity 7.12

Modified code for *FirstSprite*:

```
// Project: FirstSprite
// Created: 2015-01-16

//*** Set window title and size ***
SetWindowTitle("First Sprite")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Load image ***
LoadImage(1, "Ball.png")

//*** Create sprite ***
CreateSprite(1,1)

//*** Resize sprite ***
SetSpriteSize(1,-1,10)

//*** Display the sprite ***
Sync()

//*** Wait two seconds ***
Sleep(2000)
```

```
//*** Move the sprite across screen ***
for p = 1 to 100
  SetSpritePosition(1,p,p)
  //*** and display its position ***
  PrintC("Spite's position: ")
  PrintC("(")
  PrintC(GetSpriteX(1))
  PrintC(", ")
  PrintC(GetSpriteY(1))
  Print(")")
  Sync()
next p

do
    Sync()
loop
```

Activity 7.13

Modified code for *SpriteDepth*:

```
// Project: SpriteDepth
// Created: 2015-01-16

//*** Set window title and size ***
SetWindowTitle("Sprite Depth")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen ***
ClearScreen()

//*** Load images ***
ball_img = LoadImage("Ball.png")
poppy_img = LoadImage("Poppy.png")

//*** Create ball sprite ***
ball_spr = CreateSprite(ball_img)
SetSpriteSize(ball_spr,10,-1)

//*** Create poppy sprite on layer 9 ***
poppy_spr = CreateSprite(poppy_img)
SetSpriteSize(poppy_spr,20,-1)
SetSpritePosition(poppy_spr,40,40)
SetSpriteDepth(poppy_spr,9)

do
    //*** Move ball sprite across the screen ***
    for p = 1 to 100
      SetSpritePosition(ball_spr,p,p)
      Sync()
    next p
loop
```

Activity 7.14

To have the ball pass over the poppy all that is required is to move the poppy to a lower layer by modifying the following section of code in *SpriteDepth*:

```
//*** Create poppy sprite on layer 11 ***
poppy_spr = CreateSprite(poppy_img)
SetSpriteSize(poppy_spr,20,-1)
SetSpritePosition(poppy_spr,40,40)
SetSpriteDepth(poppy_spr,11)
```

Activity 7.15

Modified code for *SpriteDepth*:

```
// Project: SpriteDepth
// Created: 2015-01-16

//*** Set window title and size ***
SetWindowTitle("Sprite Depth")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)
ClearScreen()

//*** Load images ***
ball_img = LoadImage("Ball.png")
poppy_img = LoadImage("Poppy.png")

//*** Create ball sprite ***
ball_spr = CreateSprite(ball_img)
SetSpriteSize(ball_spr,10,-1)

//*** Create poppy sprite on layer 8 ***
```

```
poppy_spr = CreateSprite(poppy_img)
SetSpriteSize(poppy_spr,20,-1)
SetSpritePosition(poppy_spr,40,40)
SetSpriteDepth(poppy_spr,8)

//*** Create a poppy clone ***
poppy2_spr = CloneSprite(poppy_spr)
SetSpritePosition(poppy2_spr,20,20)

//*** Do nothing ***
do
  //*** Move ball sprite across the screen ***
  for p = 1 to 100
    SetSpritePosition(ball_spr,p,p)
    Sync()
  next p
loop
```

The ball appears behind both poppies.

Activity 7.16

Modified code for *SpriteDepth*:

```
// Project: SpriteDepth
// Created: 2015-01-16

//*** Set window title and size ***
SetWindowTitle("Sprite Depth")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)
ClearScreen()

//*** Load images ***
ball_img = LoadImage("Ball.png")
poppy_img = LoadImage("Poppy.png")

//*** Create ball sprite ***
ball_spr = CreateSprite(ball_img)
SetSpriteSize(ball_spr,10,-1)

//*** Create poppy sprite on layer 8 ***
poppy_spr = CreateSprite(poppy_img)
SetSpriteSize(poppy_spr,20,-1)
SetSpritePosition(poppy_spr,40,40)
SetSpriteDepth(poppy_spr,8)

//*** Create a poppy clone ***
poppy2_spr = CloneSprite(poppy_spr)
SetSpritePosition(poppy2_spr,20,20)

do
  //*** Move ball sprite across the screen ***
  for p = 1 to 100
    SetSpritePosition(ball_spr,p,p)
    Sync()
  next p
  //*** Make poppies invisible ***
  SetSpriteVisible(poppy_spr,0)
  SetSpriteVisible(poppy2_spr,0)
loop
```

Activity 7.17

No solution required.

Activity 7.18

Modified code for *RotateSprite*:

```
// Project: RotateSprite
// Created: 2015-02-26

//*** Set window size and title ***
SetWindowSize(1024,768,0)
SetWindowTitle("Rotate a Sprite")

//*** Clear the screen to grey ***
SetClearColor(200,200,200)
ClearScreen()

//*** Set up sprite ***
img = LoadImage("Arrow.png")
spr = CreateSprite(img)
SetSpriteSize(spr,20,-1)
SetSpritePosition(spr,40,50)
```

```
//*** Display rotating sprite and its angle ***
do
  for angle = 0 to 359
    SetSpriteAngle(spr,angle)
    Print(GetSpriteAngle(spr))
    Sync()
  next angle
loop
```

The sprite rotates about its centre.

Activity 7.19

Modified code for *RotateSprite*:

```
// Project: RotateSprite
// Created: 2015-02-26

//*** Set window size and title ***
SetWindowSize(1024,768,0)
SetWindowTitle("Rotate a Sprite")

//*** Clear the screen to grey ***
SetClearColor(200,200,200)
ClearScreen()

//*** Set up sprite ***
img = LoadImage("Arrow.png")
spr = CreateSprite(img)
SetSpriteSize(spr,20,-1)
SetSpritePosition(spr,40,50)

//*** Move sprite offset to centre left ***
SetSpriteOffset(spr, 0, GetSpriteHeight(spr)/2)

//*** Display rotating sprite and its angle ***
do
  for angle = 0 to 359
    SetSpriteAngle(spr,angle)
    Print(GetSpriteAngle(spr))
    Sync()
  next angle
loop
```

Activity 7.20

Modified code for *RotateSprite*:

```
// Project: RotateSprite
// Created: 2015-02-26

//*** Set window size and title ***
SetWindowSize(1024,768,0)
SetWindowTitle("Rotate a Sprite")

//*** Clear the screen to grey ***
SetClearColor(200,200,200)
ClearScreen()

//*** Set up sprite ***
img = LoadImage("Arrow.png")
spr = CreateSprite(img)
SetSpriteSize(spr,20,-1)
SetSpritePosition(spr,40,50)

//*** Move sprite offset to centre left ***
SetSpriteOffset(spr, 0, GetSpriteHeight(spr)/2)

//*** Display rotating sprite and its offset ***
do
  for angle = 0 to 359
    SetSpriteAngle(spr,angle)
    Print(GetSpriteXByOffset(spr)-GetSpriteX(spr))
    Print(GetSpriteYByOffset(spr)-GetSpriteY(spr))
    Sync()
  next angle
loop
```

Activity 7.21

No solution required.

Activity 7.22

Modified code for *UserInteraction*:

```
// Project: UserInteraction
// Created: 2015-01-17
```

```
//*** Set window title and size ***
SetWindowTitle("User Interaction")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)
ClearScreen()

do
  //*** Check for held down ***
  if GetPointerState()=1
    Print("Press held")
  else
    Print("No press")
  endif
  Sync()
loop
```

Activity 7.23

Within a window, the point(0,0) can be reached, but the other limit cannot. Testing on a Windows 7 machine gave a maximum of around (99.9, 99.9).

On a tablet, the value of the bottom right maybe greater than (100,100) because the point pressed is outside the area of the screen used by the app if it is to maintain the stated aspect ratio.

Activity 7.24

Modified code for *PointerPosition*:

```
// Project: PointerPosition
// Created: 2015-01-17

//*** Set window title and size ***
SetWindowTitle("Pointer Position")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Clear the screen to dark grey ***
SetClearColor(200,200,200)
ClearScreen()

do
  //*** If pressed, record position ***
  if GetPointerPressed() = 1
    x# = GetPointerX()
    y# = GetPointerY()
  endif
  //*** Display coordinates of latest press ***
  PrintC("(")
  PrintC(x#)
  PrintC(",")
  PrintC(y#)
  Print(")")
  Sync()
loop
```

Activity 7.25

Modified code for *SpriteOver*:

```
// Project: SpriteOver
// Created: 2015-01-17

//*** Set window title and size ***
SetWindowTitle("Sprite Over")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Set screen background to grey ***
SetClearColor(120,120,120)
ClearScreen()

//*** Load images required ***
ball_img = LoadImage("Ball.png")
poppy_img = LoadImage("Poppy.png")

//*** Create, size and position ball sprite ***
ball_spr = CreateSprite(ball_img)
SetSpriteSize(ball_spr, 10,-1)
SetSpritePosition(ball_spr, 20, 33)

//*** Create, size and position poppy sprite ***
poppy_spr = CreateSprite(poppy_img)
```

```
        SetSpriteSize(poppy_spr, 15,-1)
        SetSpritePosition(poppy_spr, 60, 30)

        do
            //*** If mouse pressed or screen touched ***
            if GetPointerPressed() = 1
                //*** Get ID of any sprite under pointer ***
                spr_hit =
                ⮡GetSpriteHit(GetPointerX(),GetPointerY())
                //*** If a sprite is under pointer, hide it ***
                if spr_hit <> 0
                    SetSpriteVisible(spr_hit,0)
                endif
            endif
            Sync()
        loop
```

Activity 7.26

To have the sprites disappear/appear as they are clicked, change the last `if` statement in *SpriteOver* from

```
//*** If a sprite is under pointer, hide it ***
if spr_hit <> 0
    SetSpriteVisible(spr_hit,0)
endif
```

to

```
//*** If a sprite is under pointer, invert it
⮡visibility it ***
if spr_hit <> 0
    SetSpriteVisible(spr_hit,1-GetSpriteVisible(
    ⮡spr_hit))
endif
```

Activity 7.27

Code for *TextAlignment*:

```
// Project: TextAlignment
// Created: 2015-08-24

//*** Set window title and size ***
SetWindowTitle("Text Alignment")
SetWindowSize(1024,720,0)
SetDisplayAspect(1024/720.0)

//*** Set screen background to grey ***
SetClearColor(120,120,120)
ClearScreen()

//*** Display the text ***
CreateText(1,"Hello world")
SetTextPosition(1,50,12)
CreateText(2,"Hello world")
SetTextAlignment(2,2)
SetTextPosition(2,50,20)
CreateText(3,"Hello world")
SetTextAlignment(3,1)
SetTextPosition(3,50,28)

do
    Sync()
loop
```

Activity 7.28

No solution required.

8

Spot the Difference Game

In this Chapter:

❑ **Designing a Game**

❑ **Game Documentation**

❑ **Designing Screen Layouts**

❑ **State Diagrams**

❑ **Incremental Builds**

❑ **Game Testing**

Game - Spot the Difference

Introduction

At last, we know enough AGK BASIC to create a first game.

The game we are going to create in this chapter is a 21st century update on the spot-the-difference game so beloved of many magazines. Two almost identical images are displayed side-by-side and the challenge for the player is to spot the differences between the two images.

There will have to be some compromises in the features included in the game and, more importantly, in the structure of our program since there is still much to learn about good program design. However, we will return to this project briefly in other chapters to correct its shortcomings.

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 supplying an in-depth description of the game, defining the rules or stating how the game is won. But these are details that should be created for any new concept game.

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 (although we will not be including sounds or music at this stage).

User interaction methods and help options are other aspects that have to be considered. How is the user to learn the rules of the game? Do we add an accompanying video tutorial, write a user manual, or include detailed help features within the game?

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 (or 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 player wins by correctly clicking on all six differences and the total time taken (in seconds) is displayed. If all six differences have not been found within 2 minutes or the player has made more than 8 clicks, then the game finishes and a display states that the player has lost and gives the reason for the defeat.

Screen Layouts

Before we start on the screen layout design, we need to decide on the screen orientation to be used by the game – portrait or landscape. In this game, because we want to have the two differing images side-by-side, our best choice is landscape, so we need to design our screen layouts accordingly.

This game will have five screen layouts: splash screen, main game screen, finish screen (Win), finish screen (Lose) and credits screen.

We 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!
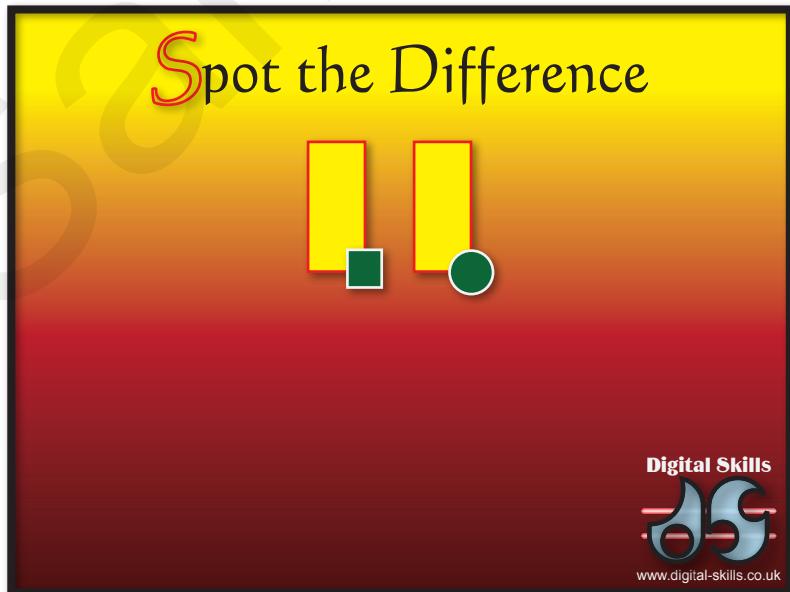
Image resolution is also important. A 1024 x 768 image will look fine on a device with the resolution of the original iPad, but it may not look so sharp on a later 2048 x 1536 screen. But, then again, higher resolution images require more memory and more processing power to move about the screen.

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 of any size and always produce the best possible quality image.

The splash screen (filename : *Splash.jpg*) is shown in FIG-8.1.

FIG-8.1

The Splash Screen

The splash screen image is held as a single JPEG image. Note that it includes the name of the game, the company name (Digital Skills), and the Digital Skills website address. Always publicise your company!

The second image (see FIG-8.2) is of the game screen containing the two photographs that form the game.

FIG-8.2

The Main Screen

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

The third image is the end screen displayed when the player wins. This shows the total time taken in seconds (see FIG-8.3).

FIG-8.3

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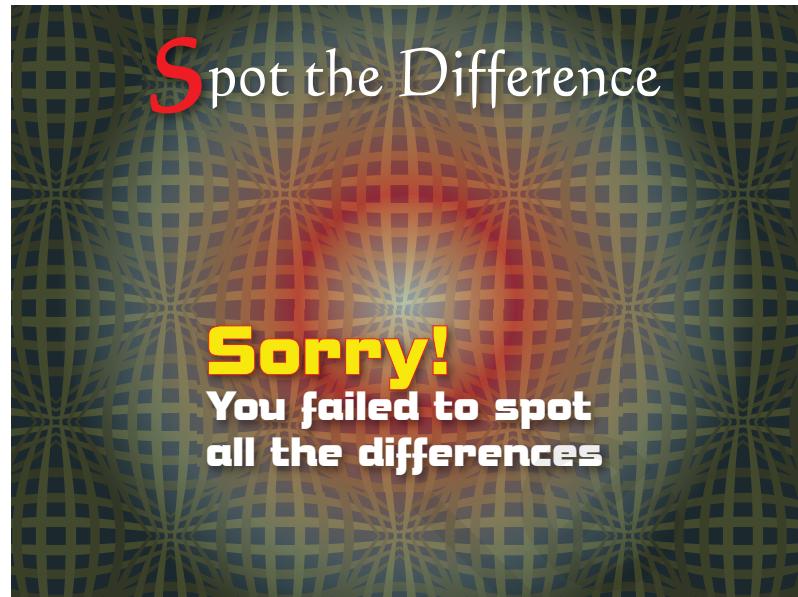
Again, you can see that a space has been left for the actual number of seconds taken to find all the differences. In addition, this screen also shows a separate button sprite in the bottom-right which allows the user to view the credits screen if required.

This screen might also show a **New Game** button, but since this game only offers a single pair of images, there's no need for a replay option.

The fourth image appears when the player loses the game – either by timing out or by clicking on the image too often (see FIG-8.4).

FIG-8.4
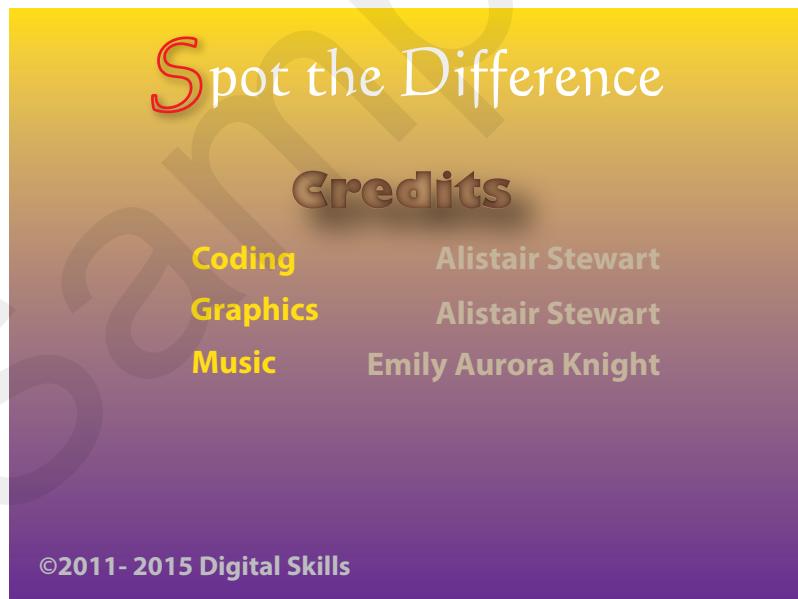
The Lose End Screen

The final image (see FIG-8.5) shows the names of those involved in creating the various aspects of the game: graphics, code, music.

FIG-8.5

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-8.6).

FIG-8.6

The Circle Spite

Other Resources
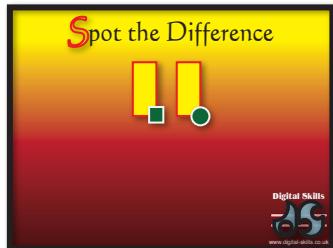
Typical other resources are sound, music and even video elements. Like the images, these have to be created. However, our game has none of these additional elements.

Overall Game Document

A useful document to produce is one showing not only the four screen layouts but
also giving details of any sounds or actions that can occur during each stage of the
game (see FIG-8.7).

Splash Screen

Main Screen

- Shows circle over each
  correctly selected difference

- Shows time game has been
  running

Win End Screen
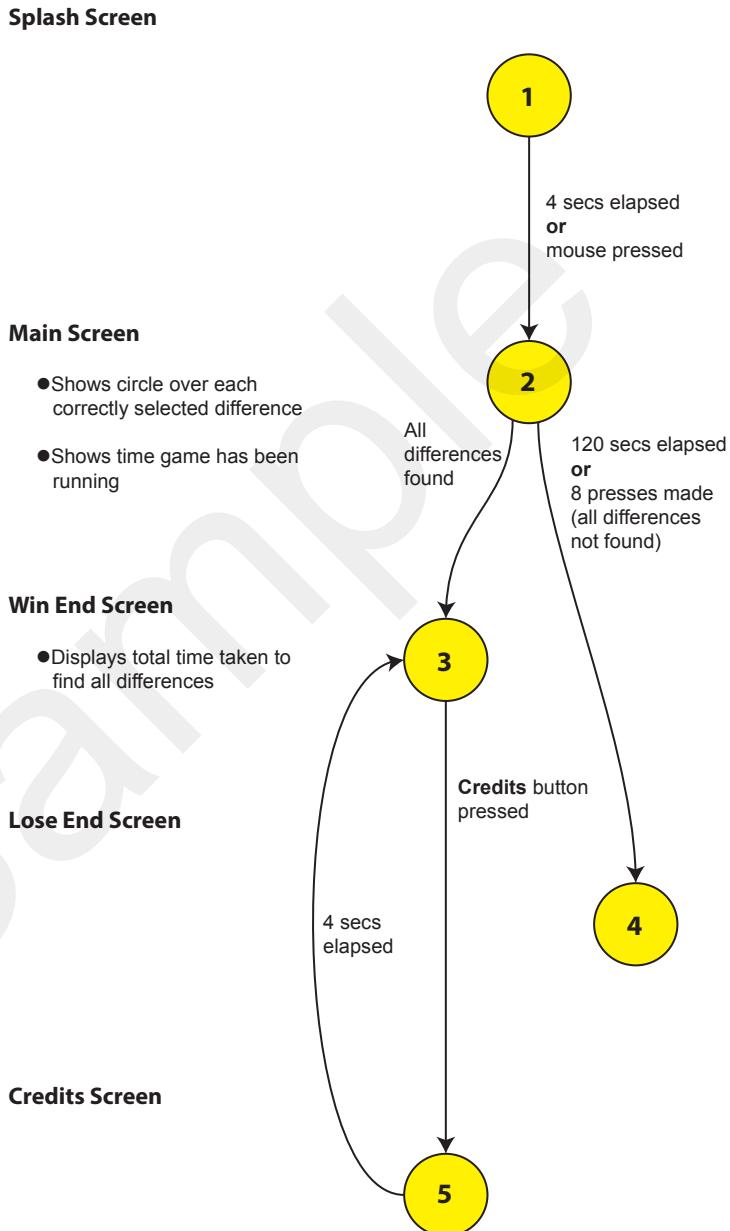
- Displays total time taken to
  find all differences

Lose End Screen

Credits Screen

4 secs elapsed
or
mouse pressed

All
differences
found

120 secs elapsed
or
8 presses made
(all differences
not found)

Credits button
pressed

4 secs
elapsed

In the *Main* and *Win 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.

On the right of FIG-8.6 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   Set window size
2   Display splash screen and start timer
3   Load resources
4   Remove splash screen
5   Set up game screen
6   Play game
7   End game

More detail can be added to some of these using stepwise refinement:

1 Set up window
    1.1 Create a window 1024 x 768
    1.2 Set window's title to "Spot The Difference Game"

2 Display splash screen and start timer
    2.1 Load splash screen image "Splash.jpg"
    2.2 Display image in sprite over full window
    2.3 Reset timer

3 Load resources
    3.1 Load main screen image "Main.jpg"
    3.2 Load finish(lose) image "FinishLose.jpg"
    3.3 Load finish(win) image "FinishWin.jpg"
    3.3 Load credits image "Credits.jpg"
    3.4 Load credits button image "CreditButton.png"
    3.5 Load circle image "Circle.png"

4 Remove splash screen
    4.1 WHILE time < 4 secs AND mouse not pressed DO
    4.2 ENDWHILE
    4.3 Delete splash screen sprite
    4.4 Delete splash screen image

5 Set up game screen
    5.2 Display Main screen
    5.3 Add circles over differences
    5.4 Hide circles

6 Play game
    6.1 Start timer display
    6.2 Set count of differences found to zero
    6.3 Set count of button presses made to zero
    6.4 REPEAT
    6.5   IF mouse button pressed THEN
    6.6   Increment button presses
    6.7   IF user selected a difference THEN
    6.8   Show appropriate circle

Game Code

The game code follows the logic given above. The first section loads and displays the splash screen image.

Structured English:

1   Set window size

Code:

```
//*******************************************
//*** Program    : Spot the Difference    ***
//*** Version    : 1.1                     ***
//*** Author     : A. Stewart              ***
//*** Date       : 3 Feb 2015              ***
//*** Language   : AGK BASIC v2.10         ***
//*** Platform   : PC Windows 7            ***
//*** Description: Displays two slightly   ***
//***              differing images. The user***
//***              has to click on the 6   ***
//***              differences within a time ***
//***              limit and using a limited ***
//***              number of clicks        ***
//*******************************************

//*** Set window properties ***
SetWindowSize(1024, 768, 0)
SetWindowTitle("Spot The Difference Game")
//*** Clear the screen ***
ClearScreen()
```

Notice that we have taken the time to add an introductory set of comments detailing various aspects of the project including the author's name, the language used, the platform that the program is being tested on and a brief description of what the program does.

Rather than wait until we have a complete program before we check if our code is correct, a much better approach is to to test each section as it is coded. This is known as an **incremental build**.

> **Activity 8.1**
>
> Start a new project called *SpotTheDifference*. Compile the default code so that the project's *media* folder is created.
>
> From the book resources you downloaded earlier, copy the *Chapter8* folder's files to *SpotTheDifference*'s *media* folder. These files are:
> *Circle.png, Credits.jpg, CreditsButton.png, FinishLose.jpg, FinishWin.jpg, Main.jpg, Splash.jpg*.
>
> Replace the code in *main.agc* with the code given above (change the author, date and other appropriate details as required).
>
> Finish the code with the lines:
> ```
> //*** Keep refreshing the screen ***
> do
>     Sync()
> loop
> ```
> Now run the program and check that a window is created.

With the first part of our structured English successfully converted to AGK BASIC, we can now move on to the next statement in our outline.

Structured English:

> 2 Display splash screen

Code:

```
//*** Display splash screen ***
splash_img = LoadImage("Splash.jpg")
splash_spr = CreateSprite(splash_img)
SetSpriteSize(splash_spr,100,100)
Sync()
//*** and reset timer ***
ResetTimer()
```

> **Activity 8.2**
>
> Add the code given above to the appropriate place in *main.agc*.
>
> Run the program and check that the splash screen appears and completely fills the window.

Structured English:

> 3 Load resources

Code:

```
//*** Load resources ***
main_img = LoadImage("Main.jpg")
finwin_img = LoadImage("FinishWin.jpg")
finlose_img = LoadImage("FinishLose.jpg")
credits_img = LoadImage("Credits.jpg")
button_img = LoadImage("CreditsButton.png")
circle_img = LoadImage("Circle.png")
```

Structured English:

    4   Remove splash screen

Code:

```
//*** Remove splash screen ***
while timer() < 4 and GetPointerPressed() = 0
    Sync()
endwhile
DeleteSprite(splash_spr)
DeleteImage(splash_img)
```

Note that we need to add a `Sync()` statement within the `while` loop. This is required not because we need the screen to be updated, but because the `Sync()` function carries out other duties including detecting mouse button clicks.

Structured English:

    5   Set up game screen

Code:

```
//*** Set up game screen ***
main_spr = CreateSprite(main_img)
SetSpriteSize(main_spr,100,-1)
//*** Load circles at image differences ***
circle_spr1 = CreateSprite(circle_img)
SetSpriteSize(circle_spr1,-1,10)
SetSpritePosition(circle_spr1,91,86)
circle_spr2 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr2,51.5,22)
circle_spr3 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr3,49,68)
circle_spr4 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr4,73,66)
circle_spr5 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr5,88.5,66)
circle_spr6 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr6,55.75,62.5)
//*** Hide the circles ***
SetSpriteVisible(circle_spr1,0)
SetSpriteVisible(circle_spr2,0)
```

```
SetSpriteVisible(circle_spr3,0)
SetSpriteVisible(circle_spr4,0)
SetSpriteVisible(circle_spr5,0)
SetSpriteVisible(circle_spr6,0)
```

Since we'll want to check that the circles are correctly positioned over the six differences, we will need to comment out the last six lines of this new code. This will keep the circles visible.

Activity 8.5

Add this latest code to the appropriate point in *main.agc*.

Comment out the last six lines and run the program to check that the circles are positioned correctly over the differences (some of the differences are quite difficult to see).

Remove the comment characters and run the program again to make sure all six circles have disappeared.

The next step in the logic is the most complicated one, so rather than use the level 1 logic, we'll code this using the level to description.

Structured English:

> 6.1 Start timer display

Code:

```
//*** Play game ***
//** Start timer display **
//* Reset the timer *
ResetTimer()
//* Set up timer text resource *
timer_txt = CreateText(str(GetSeconds()))
SetTextColor(timer_txt,0,0,0,255)
SetTextAlignment(timer_txt,2)
SetTextPosition(timer_txt,94,6)
```

Notice the use of the comments with two and one, rather than three asterisks. This is done simply to remind us that these comments are a more detailed description of parts of the main step – it's the comments' equivalent of level 2 and level 3 stepwise refinement statements.

Activity 8.6

Add this latest code to the appropriate point in *main.agc*.

Run the program and check that a zero appears close to the text *Time:* in the top-right corner.

Structured English:

> 6.2 Set count of differences found to zero
> 6.3 Set count of button presses made to zero

Code:

```
//** Set count of differences found to zero **
```

```
                    found = 0
                    //** Set count of button presses to zero **
                    clicks = 0
```

Since this won't make any difference to what is displayed by the program, we'll convert more of our structured English before updating the program code again.

Structured English:

6.4   REPEAT
6.5   IF mouse button pressed THEN
6.6   Increment button presses
6.7   IF user selected a difference THEN
6.8   Show appropriate circle
6.9   Increment differences found count
6.10   ENDIF
6.11   ENDIF
6.12   Update time
6.13   UNTIL count is 6 OR 8 presses OR time > 120 secs
6.14   Record the time

Code:

```
repeat
    //** mouse button pressed **
    if GetPointerPressed() = 1
        //** Increment button presses **
        inc clicks
        //** If user selected a difference **
        //* Get ID of sprite hit *
        hit = GetSpriteHit(GetPointerX(),GetPointerY())
        //* If sprite hit is hidden circle *
        if hit >= circle_spr1 and hit <= circle_spr6 and
        ↳GetSpriteVisible(hit) = 0
            //** Show appropriate circle **
            SetSpriteVisible(hit,1)
            //** Increment difference found count **
            inc found
        endif
    endif
    //** Update time **
    SetTextString(timer_txt,Str(GetSeconds()))
    //* Update the screen *
    Sync()
until found = 6 or clicks = 8 or GetSeconds() > 120
//** Record the time **
time_taken - GetSeconds()
```

The part of the code that checks that one of the circles has been clicked (`hit >= circle_spr1 and hit <= circle_spr6`) makes use of the fact that AGK assigns consecutive ID values to sprites. The final part of the same condition checks that the hit is on an invisible circle, thereby ensuring the user can't click twice on the same circle.

Activity 8.7

Add the last two code conversions (for 6.2 to 6.13) to the appropriate point in *main.agc*.

Run the program and check that the circles appear when clicked on.

6.14 Delete main screen resources

Code:

```
//** Delete main screen resources **
//* Delete circle sprites *
for c = circle_spr1 to circle_spr6
   DeleteSprite(c)
next c
//* Delete main screen sprite *
DeleteSprite(main_spr)
//* Delete the images used by these sprites *
DeleteImage(circle_img)
DeleteImage(main_img)
//* Delete time elapsed text resource *
DeleteText(timer_txt)
//* Update screen *
Sync()
```

Activity 8.8

Add this latest code to the appropriate point in *main.agc*.

Run the program to check that the main screen display is removed.

Like step 6 in the structured English, step 7 is complex enough to warrant being converted using the level 2 description:

Structured English:

7.1 IF all 6 differences found THEN
7.2   Show Win End Screen
7.3   Display time taken
7.4   Display Credits button
7.5   DO
7.6   IF Credits button pressed THEN
7.7   Show Credits screen for 5 seconds
7.8   ENDIF
7.9   LOOP
7.10 ELSE
7.11   Display Lose End Screen
7.12 ENDIF

Code:

```
//** If all 6 differences found **
if found = 6
   //** Show Win End Screen **
   finwin_spr = CreateSprite(finwin_img)
   SetSpriteSize(finwin_spr,100,-1)
   //** Display time taken **
   totaltime_txt = CreateText(str(time_taken))
   SetTextColor(totaltime_txt,0,0,0,255)
   SetTextSize(totaltime_txt,5.5)
   SetTextAlignment(totaltime_txt,2)
   SetTextPosition(totaltime_txt,46,57.8)
   //** Display Credits button **
   button_spr = CreateSprite(button_img)
   SetSpriteSize(button_spr,15,-1)
   SetSpritePosition(button_spr,80,90)
   do
```

```
             //** If Credits button pressed **
             if GetPointerPressed() = 1 and GetSpriteHit(GetPointerX()
             ,GetPointerY()) = button_spr
                //** Show credits screen for 5 seconds **
                   credits_spr = CreateSprite(credits_img)
                   SetSpriteSize(credits_spr,100,100)
                   //* Credits screen placed over win screen *
                   SetSpriteDepth(credits_spr,8)
                   Sync()
                   Sleep(5000)
                   //* Remove Credits screen *
                   DeleteSprite(credits_spr)
                endif
                Sync()
          loop
       else
          //** Show Lose end screen **
          finlose_spr = CreateSprite(finlose_img)
          SetSpriteSize(finlose_spr,100,100)
          do
               Sync()
          loop
       endif
```

The *Credits* screen is displayed "on top of" the *End* screen, so when it is deleted after 5 seconds, the *End* screen reappears.

> **Activity 8.9**
>
> Add this latest code to the appropriate point in *main.agc*.
>
> Run the program to check that the main screen display is removed.

Solutions

Activity 8.1

The *media* folder should contain the following files:

> *Circle.png*
> *Credits.jpg*
> *CreditsButton.png*
> *FinishLose.jpg*
> *FinishWin.jpg*
> *Main.jpg*
> *Splash.png*

The complete program code in *main.agc* is:

```
//*********************************************
//*** Program    : Spot the Difference    ***
//*** Version    : 1.1                     ***
//*** Author     : A. Stewart              ***
//*** Date       : 3 Feb 2015              ***
//*** Language   : AGK BASIC v2.10         ***
//*** Platform   : PC Windows 7            ***
//*** Description: Displays two slightly   ***
//***              differing images. The user***
//***              has to click on the 6   ***
//***              differences within a time ***
//***              limit and using a limited ***
//***              number of clicks        ***
//*********************************************


//*** Set window properties ***
SetWindowTitle("Spot The Difference Game")
SetWindowSize(1024, 768, 0)

//*** Clear the screen ***
ClearScreen()

//*** Keep refreshing the screen ***
do
   Sync()
loop
```

The window should appear (size 1024 x 768) with the text *Spot The Difference Game* in the title bar.

Activity 8.2

The program code in *main.agc* is (the initial block of comments have been removed from the remaining solutions):

```
//*** Set window properties ***
SetWindowTitle("Spot The Difference Game")
SetWindowSize(1024, 768, 0)

//*** Clear the screen ***
ClearScreen()

//*** Display splash screen ***
splash_img = LoadImage("Splash.jpg")
splash_spr = CreateSprite(splash_img)
SetSpriteSize(splash_spr,100,100)
Sync()
//*** and reset timer ***
ResetTimer()

//*** Keep refreshing the screen ***
do
   Sync()
loop
```

Running the program should display the splash screen.

Activity 8.3

The program code in *main.agc* is:

```
//*** Set window properties ***
SetWindowTitle("Spot The Difference Game")
SetWindowSize(1024, 768, 0)

//*** Clear the screen ***
ClearScreen()

//*** Display splash screen ***
splash_img = LoadImage("Splash.jpg")
splash_spr = CreateSprite(splash_img)
SetSpriteSize(splash_spr,100,100)
Sync()
//*** and reset timer ***
ResetTimer()

//*** Load resources ***
main_img = LoadImage("Main.jpg")
finwin_img = LoadImage("FinishWin.jpg")
finlose_img = LoadImage("FinishLose.jpg")
credits_img = LoadImage("Credits.jpg")
button_img = LoadImage("CreditsButton.png")
circle_img = LoadImage("Circle.png")

//*** Keep refreshing the screen ***
do
   Sync()
loop
```

Running the program should display the splash screen as before.

Activity 8.4

The program code in *main.agc* is:

```
//*** Set window properties ***
SetWindowTitle("Spot The Difference Game")
SetWindowSize(1024, 768, 0)

//*** Clear the screen ***
ClearScreen()

//*** Display splash screen ***
splash_img = LoadImage("Splash.jpg")
splash_spr = CreateSprite(splash_img)
SetSpriteSize(splash_spr,100,100)
Sync()
//*** and reset timer ***
ResetTimer()

//*** Load resources ***
main_img = LoadImage("Main.jpg")
finwin_img = LoadImage("FinishWin.jpg")
finlose_img = LoadImage("FinishLose.jpg")
credits_img = LoadImage("Credits.jpg")
button_img = LoadImage("CreditsButton.png")
circle_img = LoadImage("Circle.png")

//*** Remove splash screen ***
while timer() < 4 and GetPointerPressed() = 0
   Sync()
endwhile
DeleteSprite(splash_spr)
DeleteImage(splash_img)
//*** Keep refreshing the screen ***
do
   Sync()
loop
```

Running the program should display the splash screen which should disappear after 4 seconds or when the left mouse button is pressed.

Activity 8.5

The program code in *main.agc* is:

```
//*** Set window properties ***
SetWindowTitle("Spot The Difference Game")
SetWindowSize(1024, 768, 0)

//*** Clear the screen ***
ClearScreen()

//*** Display splash screen ***
```

```
splash_img = LoadImage("Splash.jpg")
splash_spr = CreateSprite(splash_img)
SetSpriteSize(splash_spr,100,100)
Sync()
//*** and reset timer ***
ResetTimer()

//*** Load resources ***
main_img = LoadImage("Main.jpg")
finwin_img = LoadImage("FinishWin.jpg")
finlose_img = LoadImage("FinishLose.jpg")
credits_img = LoadImage("Credits.jpg")
button_img = LoadImage("CreditsButton.png")
circle_img = LoadImage("Circle.png")

//*** Remove splash screen ***
while timer() < 4 and GetPointerPressed() = 0
   Sync()
endwhile
DeleteSprite(splash_spr)
DeleteImage(splash_img)

//*** Show main screen ***
main_spr = CreateSprite(main_img)
SetSpriteSize(main_spr,100,-1)
//*** Load circles at image differences ***
circle_spr1 = CreateSprite(circle_img)
SetSpriteSize(circle_spr1,-1,10)
SetSpritePosition(circle_spr1,91,86)
circle_spr2 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr2,51.5,22)
circle_spr3 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr3,49,68)
circle_spr4 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr4,73,66)
circle_spr5 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr5,88.5,66)
circle_spr6 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr6,55.75,62.5)
//*** Hide the circles ***
SetSpriteVisible(circle_spr1,0)
SetSpriteVisible(circle_spr2,0)
SetSpriteVisible(circle_spr3,0)
SetSpriteVisible(circle_spr4,0)
SetSpriteVisible(circle_spr5,0)
SetSpriteVisible(circle_spr6,0)

//*** Keep refreshing the screen ***
do
   Sync()
loop
```

After showing the splash screen, the main screen is displayed. With the circles still visible, the differences should be highlighted as shown below.

Activity 8.6

The program code in *main.agc* is:

```
//*** Set window properties ***
SetWindowTitle("Spot The Difference Game")
SetWindowSize(1024, 768, 0)

//*** Clear the screen ***
ClearScreen()

//*** Display splash screen ***
splash_img = LoadImage("Splash.jpg")
splash_spr = CreateSprite(splash_img)
SetSpriteSize(splash_spr,100,100)
Sync()
//*** and reset timer ***
ResetTimer()

//*** Load resources ***
main_img = LoadImage("Main.jpg")
finwin_img = LoadImage("FinishWin.jpg")
finlose_img = LoadImage("FinishLose.jpg")
credits_img = LoadImage("Credits.jpg")
button_img = LoadImage("CreditsButton.png")
circle_img = LoadImage("Circle.png")

//*** Remove splash screen ***
while timer() < 4 and GetPointerPressed() = 0
   Sync()
endwhile
DeleteSprite(splash_spr)
DeleteImage(splash_img)

//*** Show main screen ***
main_spr = CreateSprite(main_img)
SetSpriteSize(main_spr,100,-1)
//*** Load circles at image differences ***
circle_spr1 = CreateSprite(circle_img)
SetSpriteSize(circle_spr1,-1,10)
SetSpritePosition(circle_spr1,91,86)
circle_spr2 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr2,51.5,22)
circle_spr3 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr3,49,68)
circle_spr4 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr4,73,66)
circle_spr5 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr5,88.5,66)
circle_spr6 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr6,55.75,62.5)
//*** Hide the circles ***
SetSpriteVisible(circle_spr1,0)
SetSpriteVisible(circle_spr2,0)
SetSpriteVisible(circle_spr3,0)
SetSpriteVisible(circle_spr4,0)
SetSpriteVisible(circle_spr5,0)
SetSpriteVisible(circle_spr6,0)

/*** Play game ***
//** Start timer display **
//* Reset the timer *
ResetTimer()
//* Set up timer text resource *
timer_txt = CreateText(str(GetSeconds()))
SetTextColor(timer_txt,0,0,0,255)
SetTextAlignment(timer_txt,2)
SetTextPosition(timer_txt,94,6)

//*** Keep refreshing the screen ***
do
   Sync()
loop
```

The main screen should show the time elapsed value in the top-left corner. This value will be zero and remain unchanged.

Activity 8.7

The program code in *main.agc* is:

```
//*** Set window properties ***
SetWindowTitle("Spot The Difference Game")
SetWindowSize(1024, 768, 0)

//*** Clear the screen ***
ClearScreen()
```

```
//*** Display splash screen ***
splash_img = LoadImage("Splash.jpg")
splash_spr = CreateSprite(splash_img)
SetSpriteSize(splash_spr,100,100)
Sync()
//*** and reset timer ***
ResetTimer()

//*** Load resources ***
main_img = LoadImage("Main.jpg")
finwin_img = LoadImage("FinishWin.jpg")
finlose_img = LoadImage("FinishLose.jpg")
credits_img = LoadImage("Credits.jpg")
button_img = LoadImage("CreditsButton.png")
circle_img = LoadImage("Circle.png")

//*** Remove splash screen ***
while timer() < 4 and GetPointerPressed() = 0
  Sync()
endwhile
DeleteSprite(splash_spr)
DeleteImage(splash_img)

/*** Show main screen ***
main_spr = CreateSprite(main_img)
SetSpriteSize(main_spr,100,-1)
//*** Load circles at image differences ***
circle_spr1 = CreateSprite(circle_img)
SetSpriteSize(circle_spr1,-1,10)
SetSpritePosition(circle_spr1,91,86)
circle_spr2 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr2,51.5,22)
circle_spr3 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr3,49,68)
circle_spr4 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr4,73,66)
circle_spr5 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr5,88.5,66)
circle_spr6 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr6,55.75,62.5)
//*** Hide the circles ***
SetSpriteVisible(circle_spr1,0)
SetSpriteVisible(circle_spr2,0)
SetSpriteVisible(circle_spr3,0)
SetSpriteVisible(circle_spr4,0)
SetSpriteVisible(circle_spr5,0)
SetSpriteVisible(circle_spr6,0)

/*** Play game ***
//*** Start timer display **
//* Reset the timer *
ResetTimer()
//* Set up timer text resource *
timer_txt = CreateText(str(GetSeconds()))
SetTextColor(timer_txt,0,0,0,255)
SetTextAlignment(timer_txt,2)
SetTextPosition(timer_txt,94,6)

//** Set count of differences found to zero **
found = 0
//** Set mouse button presses made to zero **
clicks = 0
repeat
  //** mouse button pressed **
  if GetPointerPressed() = 1
    //** Increment button presses **
    inc clicks
    //** If user selected a difference **
    //* Get ID of sprite hit *
    hit = GetSpriteHit(GetPointerX(),
    ⮡GetPointerY())
    //* If sprite hit is hidden circle *
    if hit >= circle_spr1 and hit <=circle_spr6
    ⮡and GetSpriteVisible(hit)=0
      //** Show appropriate circle **
      SetSpriteVisible(hit,1)
      //** Increment difference found count **
      inc found
    endif
  endif
  //** Update time **
  SetTextString(timer_txt,Str(GetSeconds()))
  //* Update the screen *
  Sync()
until found = 6 or clicks = 8 or GetSeconds() > 120
//** Record the time **
time_taken = GetSeconds()
```

```
//*** Keep refreshing the screen ***
do
  Sync()
loop
```

Now the time elapsed value will change every second and when the user clicks on the changes the red circles appear.

Also, the game stops when 120 seconds has elapsed, when 8 selections have been made, or when all 6 differences are found.

Activity 8.8

The complete program code in *main.agc* is:

```
//*** Set window properties ***
SetWindowTitle("Spot The Difference Game")
SetWindowSize(1024, 768, 0)

//*** Clear the screen ***
ClearScreen()

//*** Display splash screen ***
splash_img = LoadImage("Splash.jpg")
splash_spr = CreateSprite(splash_img)
SetSpriteSize(splash_spr,100,100)
Sync()
//*** and reset timer ***
ResetTimer()

//*** Load resources ***
main_img = LoadImage("Main.jpg")
finwin_img = LoadImage("FinishWin.jpg")
finlose_img = LoadImage("FinishLose.jpg")
credits_img = LoadImage("Credits.jpg")
button_img = LoadImage("CreditsButton.png")
circle_img = LoadImage("Circle.png")

//*** Remove splash screen ***
while timer() < 4 and GetPointerPressed() = 0
  Sync()
endwhile
DeleteSprite(splash_spr)
DeleteImage(splash_img)

//*** Show main screen ***
main_spr = CreateSprite(main_img)
SetSpriteSize(main_spr,100,-1)
//*** Load circles at image differences ***
circle_spr1 = CreateSprite(circle_img)
SetSpriteSize(circle_spr1,-1,10)
SetSpritePosition(circle_spr1,91,86)
circle_spr2 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr2,51.5,22)
circle_spr3 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr3,49,68)
circle_spr4 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr4,73,66)
circle_spr5 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr5,88.5,66)
circle_spr6 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr6,55.75,62.5)
//*** Hide the circles ***
SetSpriteVisible(circle_spr1,0)
SetSpriteVisible(circle_spr2,0)
SetSpriteVisible(circle_spr3,0)
SetSpriteVisible(circle_spr4,0)
SetSpriteVisible(circle_spr5,0)
SetSpriteVisible(circle_spr6,0)

/*** Play game ***
//*** Start timer display **
//* Reset the timer *
ResetTimer()
//* Set up timer text resource *
timer_txt = CreateText(str(GetSeconds()))
SetTextColor(timer_txt,0,0,0,255)
SetTextAlignment(timer_txt,2)
SetTextPosition(timer_txt,94,6)

//** Set count of differences found to zero **
found = 0
//** Set mouse button presses made to zero **
clicks = 0
repeat
  //** mouse button pressed **
```

```
      if GetPointerPressed() = 1
        //** Increment button presses **
        inc clicks
        //** If user selected a difference **
        //* Get ID of sprite hit *
         hit = GetSpriteHit(GetPointerX(),
          ⮑GetPointerY())
         //* If sprite hit is hidden circle *
         if hit >= circle_spr1 and hit <=circle_spr6
          ⮑and GetSpriteVisible(hit)=0
             //** Show appropriate circle **
             SetSpriteVisible(hit,1)
             //** Increment difference found count **
             inc found
          endif
      endif
      //** Update time **
      SetTextString(timer_txt,Str(GetSeconds()))
      //* Update the screen *
      Sync()
until found = 6 or clicks = 8 or GetSeconds() > 120
//** Record the time **
time_taken = GetSeconds()

//** Record the time **
time_taken = GetSeconds()
//** Delete main screen resources **
//* Delete circle sprites *
for c = circle_spr1 to circle_spr6
  DeleteSprite(c)
next c
//* Delete main screen sprite *
DeleteSprite(main_spr)
//* Delete the images used by these sprites *
DeleteImage(circle_img)
DeleteImage(main_img)
//* Delete time elapsed text resource *
DeleteText(timer_txt)
//* Update screen *
Sync()

//*** Keep refreshing the screen ***
do
   Sync()
loop
```

When the game completes, all resources are destroyed and
we are left with a black window. Make sure this situation
arises for all three conditions which cause the game to finish
(120 secs, 8 clicks, 6 differences).

Activity 8.9

The complete program code in *main.agc* is:

```
//*** Set window properties ***
SetWindowTitle("Spot The Difference Game")
SetWindowSize(1024, 768, 0)

//*** Clear the screen ***
ClearScreen()

//*** Display splash screen ***
splash_img = LoadImage("Splash.jpg")
splash_spr = CreateSprite(splash_img)
SetSpriteSize(splash_spr,100,100)
Sync()
//*** and reset timer ***
ResetTimer()

//*** Load resources ***
main_img = LoadImage("Main.jpg")
finwin_img = LoadImage("FinishWin.jpg")
finlose_img = LoadImage("FinishLose.jpg")
credits_img = LoadImage("Credits.jpg")
button_img = LoadImage("CreditsButton.png")
circle_img = LoadImage("Circle.png")

//*** Remove splash screen ***
while timer() < 4 and GetPointerPressed() = 0
   Sync()
endwhile
DeleteSprite(splash_spr)
DeleteImage(splash_img)

//*** Show main screen ***
main_spr = CreateSprite(main_img)
SetSpriteSize(main_spr,100,-1)
```

```
//*** Load circles at image differences ***
circle_spr1 = CreateSprite(circle_img)
SetSpriteSize(circle_spr1,-1,10)
SetSpritePosition(circle_spr1,91,86)
circle_spr2 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr2,51.5,22)
circle_spr3 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr3,49,68)
circle_spr4 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr4,73,66)
circle_spr5 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr5,88.5,66)
circle_spr6 = CloneSprite(circle_spr1)
SetSpritePosition(circle_spr6,55.75,62.5)
//*** Hide the circles ***
SetSpriteVisible(circle_spr1,0)
SetSpriteVisible(circle_spr2,0)
SetSpriteVisible(circle_spr3,0)
SetSpriteVisible(circle_spr4,0)
SetSpriteVisible(circle_spr5,0)
SetSpriteVisible(circle_spr6,0)

/*** Play game ***
//** Start timer display **
//* Reset the timer *
ResetTimer()
//* Set up timer text resource *
timer_txt = CreateText(str(GetSeconds()))
SetTextColor(timer_txt,0,0,0,255)
SetTextAlignment(timer_txt,2)
SetTextPosition(timer_txt,94,6)

//** Set count of differences found to zero **
found = 0
//** Set mouse button presses made to zero **
clicks = 0
repeat
   //** mouse button pressed **
   if GetPointerPressed() = 1
     //** Increment button presses **
     inc clicks
     //** If user selected a difference **
     //* Get ID of sprite hit *
      hit = GetSpriteHit(GetPointerX(),
       ⮑GetPointerY())
      //* If sprite hit is hidden circle *
      if hit >= circle_spr1 and hit <=circle_spr6
       ⮑and GetSpriteVisible(hit)=0
          //** Show appropriate circle **
          SetSpriteVisible(hit,1)
          //** Increment difference found count **
          inc found
       endif
   endif
   //** Update time **
   SetTextString(timer_txt,Str(GetSeconds()))
   //* Update the screen *
   Sync()
until found = 6 or clicks = 8 or GetSeconds() > 120
//** Record the time **
time_taken = GetSeconds()

//** Record the time **
time_taken = GetSeconds()
//** Delete main screen resources **
//* Delete circle sprites *
for c = circle_spr1 to circle_spr6
  DeleteSprite(c)
next c
//* Delete main screen sprite *
DeleteSprite(main_spr)
//* Delete the images used by these sprites *
DeleteImage(circle_img)
DeleteImage(main_img)
//* Delete time elapsed text resource *
DeleteText(timer_txt)
//* Update screen *
Sync()

//*** End game ***
//** If all 6 differences found **
if found = 6
   //** Show Win End Screen **
   finwin_spr = CreateSprite(finwin_img)
   SetSpriteSize(finwin_spr,100,-1)
   //** Display time taken **
   totaltime_txt = CreateText(str(time_taken))
   SetTextColor(totaltime_txt,0,0,0,255)
   SetTextSize(totaltime_txt,5.5)
   SetTextAlignment(totaltime_txt,2)
```

```
        SetTextPosition(totaltime_txt,46,57.8)
        //** Display Credits button **
        button_spr = CreateSprite(button_img)
        SetSpriteSize(button_spr,15,-1)
        SetSpritePosition(button_spr,80,90)
        do
          //** If Credits button pressed **
          if GetPointerPressed() = 1 and GetSpriteHit(
          ↳GetPointerX(),GetPointerY()) = button_spr
            //** Show credits screen for 5 seconds **
                credits_spr = CreateSprite(credits_img)
                SetSpriteSize(credits_spr,100,100)
                //* Credits screen placed over win
                ↳screen *
                SetSpriteDepth(credits_spr,8)
                Sync()
                Sleep(5000)
                //* Remove Credits screen *
                DeleteSprite(credits_spr)
            endif
            Sync()
        loop
    else
      //** Show Lose end screen **
      finlose_spr = CreateSprite(finlose_img)
      SetSpriteSize(finlose_spr,100,100)
      do
          Sync()
      loop
    endif
```

The game is now complete. Check that the two end screens appear under the appropriate conditions and that the Credits screen can be accessed from the winning screen.