5 Iteration

In this Chapter:

- ❑ **while..endwhile** Structure
- ❑ **repeat..until** Structure
- ❑ **for..next** Structure
- ❑ **do..loop** Structure
- ❑ Validating Input
- ❑ The **exit** Statement
- ❑ Testing Loop Structures

Iteration

Introduction

Iteration is the term used when one or more statements are carried out repeatedly. As we saw in Chapter 1, structured English has three distinct iterative structures: FOR .. ENDFOR, REPEAT .. UNTIL and WHILE .. ENDWHILE.

AGK BASIC, on the other hand, has four iterative structures. One of these takes the same form as their structured English equivalent, but others differ slightly and therefore care should be taken when translating structured English statements to AGK BASIC.

The `while..endwhile` Construct

The `while` statement is probably the easiest of AGK BASIC's loop structures to understand, since it is identical in operation and syntax to the WHILE loop in structured English.

This structure allows us to continually execute a section of code as long as a specified condition is being met. For example, if, in a game, a player's character sustains damage of 10 points while he stands on a "bad health" area, this can be described in structured English as

```
WHILE player on "bad health" area DO
    Reduce player's health by 10
ENDWHILE
```

which can be coded in AGK BASIC as:

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

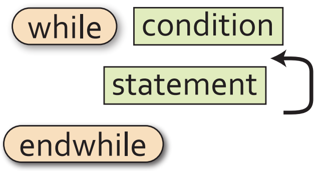
```
while floor_area = 25
   health = health - 10
endwhile
```

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

FIG-5.1

while..endwhile

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



where:

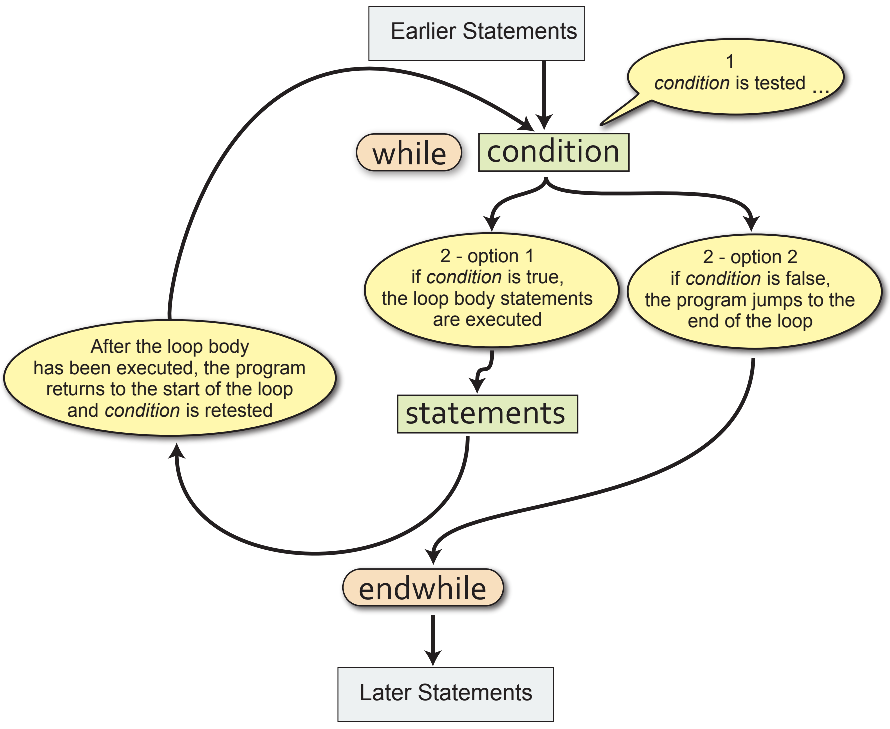
condition	is a Boolean expression and may include `and`, `or`, `not` and parentheses as required.
statement	is any valid AGK BASIC statement.

`while..endwhile` is an **entry-controlled** loop. That is, the condition at the start of the loop is tested and only if that condition is true, are the statements within the loop executed. When the `endwhile` term is reached, control returns to the `while` line and the condition is retested. If the condition is found to be false, then looping stops with an immediate jump from the `while` line to the `endwhile` line, skipping the statements in between.

A visual representation of how this loop operates is shown in FIG-5.2.

FIG-5.2

How **while.. endwhile** Operates


Earlier Statements
1 *condition* is tested ...
while
condition
2 - option 1 if *condition* is true, the loop body statements are executed
2 - option 2 if *condition* is false, the program jumps to the end of the loop
After the loop body has been executed, the program returns to the start of the loop and *condition* is retested
statements
endwhile
Later Statements

Note that the loop body may never be executed if *condition* is false when first tested.

A common use for this loop statement is validation of input. So, for example, in our number guessing game, we might ensure that the user types in a value between 0 and 9 when entering their guess by using the logic

```
Get guess
WHILE guess outside the range 0 to 9 DO
    Display error message
    Get guess
ENDWHILE
```

which can be coded in AGK BASIC using our *GetButtonEntry()* function as:

```
Print("Enter your guess (0 - 9) : ")
Sync()
Sleep(2000)
guess = GetButtonEntry()
while guess < 0 or guess > 9
   Print("Your guess must be between 0 and 9")
   Print("Enter your guess again(0 - 9) : ")
   Sync()
   Sleep(2000)
   guess = GetButtonEntry()
endwhile
```

The test `guess < 0` is not required since the function `GetButtonEntry()` does not allow negative values to be entered. However, the condition has been included so that, should `GetButtonEntry()` ever be modified to allow entry of negative values, the `while` loop will catch any values less than zero.

Activity 5.1

Modify your *Dice* project to incorporate the code given above. Check that the program works correctly by attempting to make guesses which are outside the range 0 to 9. Resave your project.

Activity 5.2

A simple dice game involves counting how many times in a row a pair of dice can be thrown to produce a value of 8 or less. The game stops as soon as a value greater than 8 is thrown.

Create a new project, *DiceCount*, which implements the following logic:

```
Set count to zero
Throw the two dice
Display dice values
WHILE the sum of the two dice <= 8 DO
    Add 1 to count
    Throw the two dice
    Display dice values
ENDWHILE
Display “You had a run of “ , count, “throws”
```

Test and save your program.

The `repeat..until` Construct

Like structured English, AGK BASIC has a `repeat..until` statement. The two structures are identical. Hence, if in structured English we write

```
Set total to zero
REPEAT
    Get a number
    Add number to total
UNTIL number is zero
```

then the same logic would be coded in AGK BASIC as

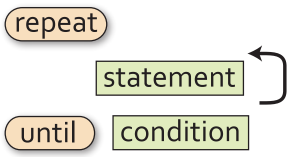
The code assumes we are using the Button routines introduced in the previous chapter to accept input.

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

The `repeat..until` statement is an **exit-controlled** loop structure. That is, the action within the loop is executed and then an exit condition is tested. If that condition is found to be true, then looping stops, otherwise the statements specified within the loop are executed again. Iteration continues until the exit condition is true. The syntax of the REPEAT statement is shown in FIG-5.3.

FIG-5.3

repeat..until



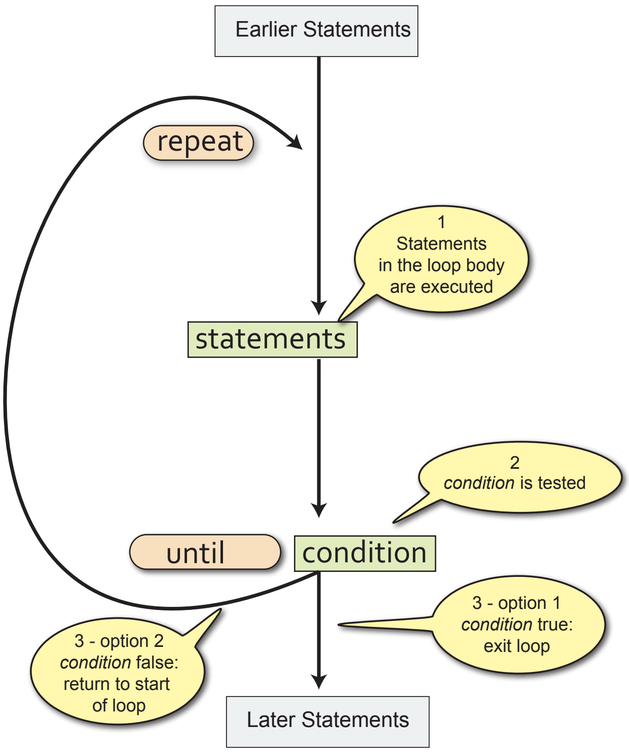
where:

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

The operation of the `repeat .. until` construct is shown graphically in FIG-5.4.

FIG-5.4

How **repeat..until** Operates


Earlier Statements
repeat
1 Statements in the loop body are executed
statements
2 *condition* is tested
until
condition
3 - option 1 *condition* true: exit loop
3 - option 2 *condition* false: return to start of loop
Later Statements

Activity 5.3

Create a new project, *Total*, to read in a series of integer values, stopping only when a zero is entered. The values entered should be totalled and that total displayed at the end of the program. Use the Buttons routines to accept input.

Use the following logic:

```
Set total to zero
REPEAT
    Get a number
    Add number to total
UNTIL number is zero
Display total
```

Test and save your project.

Activity 5.4

Modify *Dice* to allow the player to keep guessing until the correct number is arrived at.

Test and save your project.

The `for..next` Construct

In structured English, the FOR loop is used to perform an action a specific number of times. For example, we might describe dealing seven cards to a player using the logic:

```
FOR 7 times DO
    Deal card
ENDFOR
```

Sometimes the number of times the action is to be carried out is less explicit. For example, if each player in a game is to pay a £10 fine, we could write:

```
FOR each player DO
    Pay £10 fine
ENDFOR
```

However, in both examples, the action specified between the FOR and ENDFOR terms will be executed a known number of times.

In AGK BASIC the `for` construct makes use of a variable to keep a count of how often the loop is executed and the first line of the structure takes the form:

```
for variable = start_value to finish_value
```

Hence, if we want a `for` loop to iterate 7 times we would write

```
for c = 1 to 7
```

In this case *c* would be assigned the value 1 when the `for` loop is about to start. Each time the statements within the loop are completed, *c* will be incremented, and eventually, when *c* is equal to 7 and the loop body has been executed, iteration stops.

The variable used in a `for` loop is known as the **loop counter**.

Activity 5.5

Write the first line of a `for` loop that is to be executed 10 times, using a variable *j* as the loop counter. The starting value of *j* should be 1.

While structured English marks the end of a FOR loop using the term ENDFOR, in AGK BASIC the end of the loop is indicated by the term `next` followed by the name of the loop counter variable used in the `for` statement. For example, the code

```
for k = 1 to 10
   Print("*")
next k
Sync()
```

contains a single statement within the loop body and will display a column of 10 asterisks.

Activity 5.6

What would be displayed by the code

```
for p = 1 to 10
      Print(p)
next p
Sync()
```

The loop counter in a `for` loop can be made to start and finish at any value, so it is quite valid to start a loop with the line:

```
for m = 3 to 12
```

The loop counter *m* will contain the value 3 when the loop is first executed and 12 when the final execution is complete. The loop will be executed exactly 10 times.

If the start and finish values are identical, as in the line

```
for r = 10 to 10
```

then the loop is executed once only.

Where the start value is greater than the finish value, the loop will not be executed at all so the code within the loop body will be ignored. Such a result would be produced from the line

```
for k = 10 to 9
```

Normally, 1 is added to the loop counter each time the loop body is performed. However, we can change this by adding a `step` value to the `for` loop as in the example shown below:

```
for c = 2 to 10 step 2
```

In this last example the loop counter, *c*, will start at 2 and then increment to 4 on the next iteration. The program in FIG-5.5 uses the `step` option to display the 7 times table from 1 x 7 to 12 x 7.

FIG-5.5

7 Times Table

```
rem *** 7 Times Table ***

rem *** Display title ***
Print("7 Times Table")
Print("")
rem *** Display the table values ***
for c = 7 to 84 step 7
    Print(c)
next c
Sync()
do
loop
```

Activity 5.7

Start a new project, *Tables*, that implements the code shown in FIG-5.5.

Test the program.

Modify the program so that it displays the 12 times table from 1 x 12 to 12 x 12.

By using the `step` keyword with a negative value, it is even possible to create a `for` loop that reduces the loop counter on each iteration as in the line:

```
for d = 10 to 0 step -1
```

This last example causes the loop counter to start at 10 and finish at 0.

Activity 5.8

Modify *Tables* so that the 12 times table is displayed with the highest value first. That is, starting with 144 and finishing with 12.

It is possible that the `step` value given may cause the loop counter never to match the finish value. For example, in the line

```
for c = 1 to 12 step 5
```

the variable *c* will take on the values 1, 6, and 11. The loop body will not be executed when the loop counter passes the finishing value (12, in this case) and the looping will stop.

The start, finish and even step values of a `for` loop can be defined using a variable or arithmetic expression, as well as a constant. For example, in FIG-5.6 below the user is allowed to enter the upper limit of the `for` loop.

FIG-5.6

Using a Variable in a **for..next** Statement

```
#include "Buttons.agc"

SetUpButtons()
rem *** Get a number ***
Print("Enter upper limit")
Sync()
Sleep(2000)
num = GetButtonEnrty()
rem *** Display values between 1 and num ***
for c = 1 to num
   Print(c)
   Sync()
   Sleep(200)
next c
do
loop
```

The program will display every integer value between 1 and the number entered by the user. If this involves many numbers being displayed, there will not be space within the app window to show them all at the same time. Therefore, the program displays one number at a time with 0.2 secs delay between each value.

Activity 5.9

Start a new project, *OneTo*, containing the code given in FIG-5.6. (Remember you have to include the three *Buttons* files in your project folder).

Modify the program so that the user may also specify the starting value of the `for` loop.

Change the program a second time so that the user can specify a step size for the `for` loop.

Test each version of the program.

The `for` loop counter can also be specified as a real value with a `step` value which is not a whole number. For example:

```
for ch# = 1.0 to 2.0 step 0.1
   Print(ch#)
next ch#
Sync()
```

Activity 5.10

Create a project, *ForReal*, which includes the code given above and check out the result.

➽ The latest version of AGK no longer displays values to 11 decimal places; only 6, so the rounding errors are no longer visible but still occur internally.

Notice that most of the values displayed by the last Activity are slightly out. For example, instead of the second value displayed being 1.1, it displays as 1.10000002384.

This difference is caused by rounding errors when converting from the decimal values that we use to the binary values favoured by the computer.

Although we might have expected the `for` loop to perform 11 times (1.0,1.1,1.2, etc. to 2.0), in fact, it only performs 10 times up to 1.90000021458. Again, this discrepancy is caused by the rounding error problem.

Activity 5.11

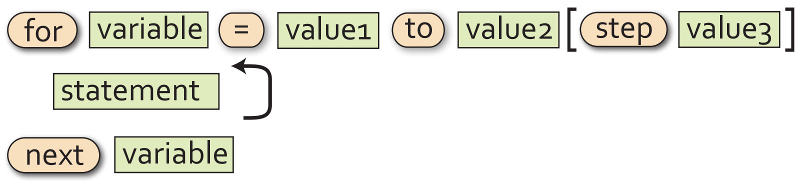
Modify *ForReal* so that the upper limit of the loop is 2.01.

How many times is the iteration performed now?

The format of the `for..next` construct is shown in FIG-5.7.

FIG-5.7

for..next



where:

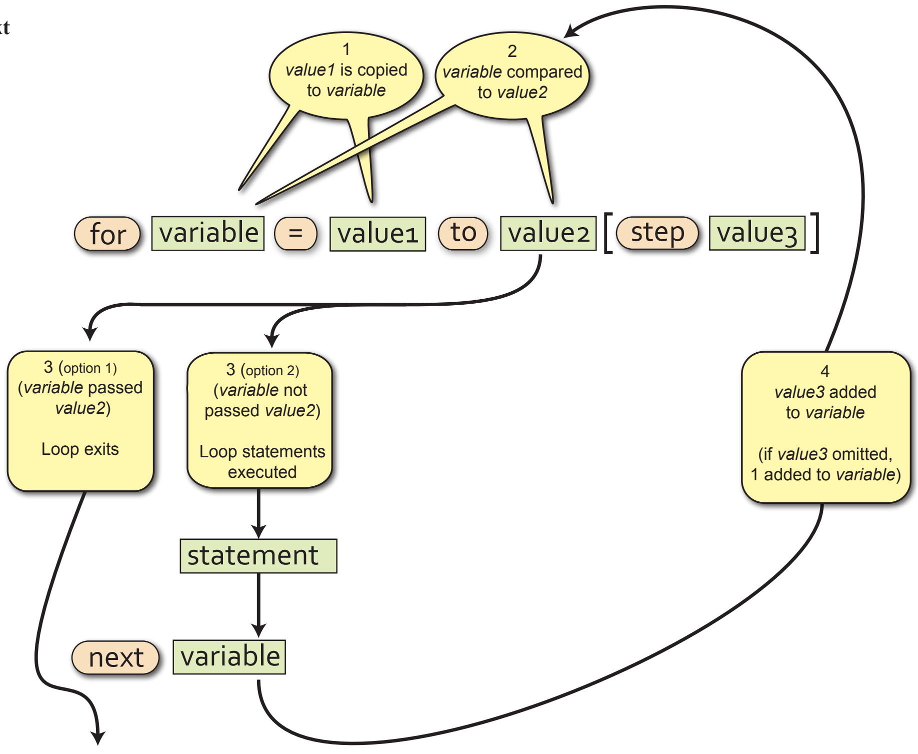
variable is either an integer or real variable. Both *variable* tiles in the diagram refer to the same variable. Hence, the name used after the keywords `for` and `next` must be the same. This variable is known as the **loop counter**.

value1	is the initial value of the loop counter. The loop counter will contain this value the first time the statements within the loop are executed.
value2	is the final value of the loop variable. The loop variable will usually contain this value the last time the loop body is executed.
value3	is the value to be added to the loop counter after each iteration. If this is omitted then a value of 1 is added to the loop counter.
statement	is any valid AKG BASIC statement.

The operation of the `for..next` statement is shown graphically in FIG-5.8.

FIG-5.8

How **for..next** Operates



1 *value1* is copied to *variable*
2 *variable* compared to *value2*
for
variable
=
value1
to
value2
[step value3]
3 (option 1) (*variable* passed *value2*) Loop exits
3 (option 2) (*variable* not passed *value2*) Loop statements executed
4 *value3* added to *variable* (if *value3* omitted, 1 added to *variable*)
statement
next
variable

Activity 5.12

Create a new project, *InTotal*, which reads in and displays the total of 6 numbers. Make use of the *Buttons* files for input.

Test and save your project.

Activity 5.13

Start a new project called *Shades*.

Code a program which uses a `for` loop with a start value of 0 and finish of 255.

Inside the loop, execute a `SetClearColor()` statement and use the value of the loop counter as the red parameter to the statement. The green and blue parameter values for the `SetClearColor()` statement should both be zero.

Add a delay (using `Sleep()`) of 20 milliseconds between each iteration of the loop.

Test and save your project.

Finding the Smallest Value in a List of Values

There are several tasks that will crop up over and over again in your programs. One of these is finding the smallest value in a list of numbers. This is a trivial enough task for our own brains as long as the list is short enough to be taken in at a glance, but if asked how you managed to come up with the correct answer, you might struggle to give a verbal description of the strategy you used.

Now, let's imagine you wanted to record the coldest temperature achieved in your area during the current year. Since this involves a longer list of data which also takes a full year to access, you would have to come up with an organised way of getting the information you want. Perhaps you would write down the lowest temperature on January 1st and then check each day to see if a lower temperature has been achieved. When a lower temperature does occur, you can erase the previous record and write down this new temperature. By the end of the year your record would show the lowest temperature achieved during the year.

This is exactly how we tackle the same type of problem in a computer program. We set up one variable to hold the smallest value we've come across so far and if a later value is smaller, it is copied into this variable. The algorithm used is given below and assumes 7 numbers will be entered in total:

```
Get first number
Set smallest to first number
FOR 6 times DO
    Get next number
    IF number < smallest THEN
        Set smallest to number
    ENDIF
ENDFOR
Display smallest
```

Activity 5.14

Create a new project called *Smallest*.

In this program implement the logic shown above to display the smallest of 5 integer values entered.

Modify the program to find the largest, rather than the smallest, of the numbers entered. Save your project.

The exit Statement

The **exit** statement is used to terminate the loop currently being executed. The next statement to be executed after an **exit** command is the statement immediately after the end of the loop. The **exit** statement takes the form shown in FIG-5.9.

FIG-5.9

The **exit** Statement

Normally, the **exit** statement will appear within an **if** statement.

Let's look at an example where the **exit** statement might come in useful. In a dice game we are allowed to throw a pair of dice 5 times and our score is the total of the five throws. However, if during our throws we throw a 1, then, according to the rules of the game, our turn ends and our final score becomes the total achieved up to that point (excluding the throw containing a 1). We could code this game as shown in FIG-5.10.

FIG-5.10

Using **exit**

```
rem *** set total to zero ***
total = 0
rem *** for 5 times do ***
for c = 1 to 5
      rem *** Display roll number ***
      PrintC(“Roll number “)
      Print(c)
      Sync()
      Sleep(1000)
      rem *** throw both dice ***
      dice1 = Random(1,6)
      dice2 = Random(1,6)
      rem *** display throw number and dice values ***
      PrintC(“dice 1 : “)
      PrintC(dice1)
      PrintC(”        dice 2 : “)
      Print(dice2)
      Sync()
      Sleep(4000)
      rem *** if either dice is a 1 then quit loop ***
      if dice1 = 1 or dice2 = 1
         exit
      endif
      rem *** add dice throws to total ***
      total = total + dice1 + dice2
   next c
   rem *** display final score ***
   PrintC(“your final score was : ”)
   Print(total)
   Sync()
   do
   loop
```

Activity 5.15

Create a new project call *SumDice*. Delete the existing code in *main.agc* and enter the program given in FIG-5.10.

Run the program and check that the loop exits if a 1 is thrown.

Modify the program to exit only if both dice show a 1.

The do .. loop Construct

The `do..loop` construct is a rather strange loop structure, since, while other loops are designed to terminate eventually, the `do .. loop` structure will continue to repeat the code within its loop body indefinitely.

The default code that exists when we begin a new project makes use of this loop structure to continually display the words *Hello world* - the traditional text for a first program.

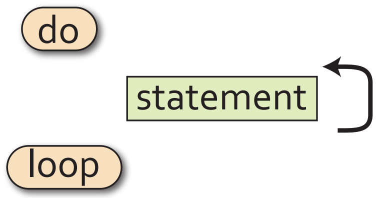
When a `do` loop is executing, then, under normal circumstances, the program will only terminate when forced to do so by an external event. In all our projects so far, the external event has been the operating system closing down our program in response to our clicking on the X button at the top-right of the app window. Alternatively, an `exit` statement can be included within the loop to allow the loop to be exited when a given condition occurs.

As we write more complex programs you will begin to understand why a `do` loop is so often needed to get the game to run smoothly.

The `do..loop` structure takes the format shown in FIG-5.11.

FIG-5.11

do..loop



Nested Loops

A common requirement within a program is to place one loop control structure within another. This is known as **nested loops**. For example, to input six game scores (each between 0 and 100) and then calculate their average, the logic required is:

```
1.  Set total to zero
2.  FOR 6 times DO
3.      Get valid score
4.      Add score to total
5.  ENDFOR
6.  Calculate average as total / 6
7.  Display average
```

This appears to have only a single loop structure beginning at statement 2 and ending at statement 5. However, if we add detail to statement 3, this gives us

```
3. Get valid score
    3.1 Read score
    3.2 WHILE score is invalid DO
    3.3     Display “Score must be between 0 to 100”
    3.4     Read score
    3.5 ENDWHILE
```

which, if placed in the original solution, results in a nested loop structure, where a `while` loop appears inside a `for` loop:

```
1.  Set total to zero
2.  FOR 6 times DO
3.1     Read score
```

```
3.2       WHILE score is invalid DO
3.3             Display "Score must be between 0 to 100"
3.4             Read score
3.5       ENDWHILE
4.        Add score to total
5.   ENDFOR
6.   Calculate average as total / 6
7.   Display average
```

Activity 5.16

Turn the above algorithm into an AKG BASIC project, *AverageScore*, using the *Buttons* files to allow input.

Run and test the program, making sure it operates as expected.

Nested for Loops

A very common example of nested loops are nested `for` loops. And, although someone new to programming can sometimes have difficulties with the concept, its actually easy enough to see real world examples of how nested `for` loops work.

Next time you are out in the car, have a look at the odometer (that's the one that tells you how many miles/kilometres the car has done). Now, look at the right two digits of the odometer. As you travel along you'll see the far right hand digit move slowly until it reaches 9; at that point it returns to zero and the digit to its left increments before the whole process repeats itself. You'll see the same sort of thing on a digital clock.

The code in FIG-5.12 emulates those last two digits on the odometer. Initially, they are set to 00 and then move onto 01, 02 ... 09,10,11, etc

FIG-5.12

Nested **for** loops

```
Rem *** Nested for loop ***
for tens = 0 to 9
    for units = 0 to 9
        PrintC(tens)
        PrintC(" ")
        Print(units)
        Sync()
        Sleep(200)
    next units
next tens
do
loop
```

The *tens* loop is known as the **outer loop**, while the *units* loop is known as the **inner loop**.

A few points to note about nested `for` loops:

- ➢ The inner loop increments fastest.
- ➢ Only when the inner loop is complete does the outer loop variable increment.
- ➢ The inner loop counter is reset to its starting value each time the outer loop counter is incremented.

Activity 5.17

Start a new project, *NestedFor*, and code the program to match FIG-5.12. Test and save your project.

Activity 5.18

What would be output by the following code?

```
for no1 = -2 to 1
      for no2 = 0 to 3
            PrintC(no1)
            PrintC(" ")
            Print(no2)
            Sync()
            Sleep(200)
      next no2
next no1
```

Testing Iterative Code

We need a test strategy when looking for errors in iterative code. Where possible, it is best to create at least three sets of values:

- ➢ Test data that causes the loop to execute zero times.
- ➢ Test data that causes the loop to execute once.
- ➢ Test data that causes the loop to execute multiple times.

For example, in *Dice* we added statements to ensure that the guess entered was in the range 0 to 9 using the following code:

```
guess = GetButtonEntry()
while guess < 0 or guess > 9
   Print("Your guess must be between 0 and 9")
   Print("Enter your guess again(0 - 9) : ")
   Sync()
   Sleep(2000)
   guess = GetButtonEntry()
endwhile
```

To test the `while` loop in this code we could use the test data shown in FIG-5.13.

FIG-5.13

Test Data

Test No.	guess
1	7
2	10, 5
3	18, 12, 3

The `while` loop is only executed if *guess* is outside the range 0 to 9, so Test 1, which uses a value inside that range, will skip the `while` loop body giving zero iterations.

Test 2 starts with an invalid value (10) for *guess*, causing the `while` loop body to be executed, and then uses a valid value (5). This loop is therefore exited after only one iteration.

Test 3 uses two invalid values (18 and 12) before entering a valid value (3), causing the `while` loop body to execute twice.

Activity 5.19

The following code is meant to calculate the average of a sequence of numbers. The sequence ends when the value zero is entered. This terminating zero is not considered to be one of the numbers in the sequence.

```
total = 0
count = 0
Print("Enter number (0 to stop)")
Sync()
Sleep(2000)
num = GetButtonEntry()
while num <> 0
      total = total + num
      count = count + 1
      Print("Enter number (0 to stop)")
      Sync()
      Sleep(2000)
      num = GetButtonEntry()
endwhile
average = total / count
PrintC("Average is ")
Print(average)
Sync()
do
loop
```

Make up a set of test values (similar in construct to FIG-5.13) for the `while` loop in the code.

Create a new project, *Average*, containing the code given above and use the test data to find out if the code functions correctly.

There will be cases where using all three tests strategies are not possible. For example, a `repeat` loop cannot execute zero times and, in this case, we have to satisfy ourselves with single and multiple iteration tests.

A `for` loop, when written for a fixed number of iterations can only be tested for that number of iterations. So a loop beginning with the line

```
for c = 1 to 10
```

can only be tested for multiple iterations (10 iterations, in this case), the exception being if the loop body contains an `exit` statement, in which case zero and one iteration tests may also be possible by supplying values which cause the `exit` statement to be terminated during the required iteration.

A `for` loop which is coded with a variable upper limit as in

```
for c = 1 to max
```

may be fully tested by making sure *max* has the values 0, 1, and more than 1 during testing.

A `do` loop can only be tested for zero and one iterations 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` loops counters can be integer or real.
- ➢ The start, finish and step values in a `for` loop can be defined using variables or arithmetic expressions.
- ➢ If the start value is equal to the finish value, a `for` loop will execute only once.
- ➢ If the start value is greater than the finish value and the `step` size is a positive value, a `for` loop will execute zero times.
- ➢ Using the `do..loop` structure creates an infinite loop.
- ➢ The `exit` statement can be used to exit from any loop.
- ➢ One loop structure can be placed within another loop structure. Such a structure is known as a nested loop.
- ➢ Loops should be tested by creating test data for zero, one and multiple iterations during execution whenever possible.

Solutions

Activity 5.1

Modified code for *Dice*:

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
rem *** Display prompt ***
Print("Guess what my number is ")
Sync()
Sleep(2000)
rem *** Get a value ***
guess = GetButtonEntry()
while guess < 0 or guess > 9
     Print("your guess must be between 0 and 9")
     Print("Enter your guess again(0 - 9) : ")
     Sync()
     Sleep(2000)
     guess = GetButtonEntry()
   endwhile
rem *** Display message ***
diff = dice - guess
if diff > 2
    Print("You guess is too low")
else
    if diff > 0
        Print("Your guess is slightly too low ")
    else
        if diff = 0
            Print("Correct")
        else
            if guess > -2
                Print("Your guess is slightly too
                 ↳high")
            else
                Print("Your guess is too high")
            endif
        endif
    endif
endif
rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
Print(guess)
Sync()
do
loop
```

Activity 5.2

Code for *DiceCount*:

```
rem *** Count dice run ***

rem *** Set count to zero ***
count = 0
rem *** Throw dice ***
dice1 = Random(1,6)
dice2 = Random(1,6)
rem *** display dice values ***
PrintC(dice1)
PrintC(" ")
Print(dice2)
Sync()
Sleep(500)
rem *** Keep going while total is less than 9 ***
while dice1 + dice2 <= 8
    rem *** add 1 to count ***
    count = count + 1
    rem *** Throw dice ***
    dice1 = Random(1,6)
    dice2 = Random(1,6)
    rem *** display dice values ***
    PrintC(dice1)
    PrintC(" ")
    Print(dice2)
    Sync()
    Sleep(500)
endwhile
PrintC("You had a run of ")
PrintC(count)
Print(" throws")
Sync()
do
loop
```

Activity 5.3

Set the app window dimensions to 768 wide by 1024 high.

Code for *Total*:

```
rem *** Total a sequence of numbers ***

rem *** include Buttons routines ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()
rem *** Set total to zero ***
total = 0
rem *** Keep going until zero entered ***
repeat
    rem *** Get value ***
    no = GetButtonEntry()
    rem *** Add value to total ***
    total = total + no
until no = 0
rem *** Display total ***
PrintC("Total = ")
Print(total)
Sync()
do
loop
```

Activity 5.4

Modified code for *Dice* (remember to indent all the code between the `repeat` and `until` terms):

```
rem *** Dice program ***
rem *** Simulates the roll of a 10-sided dice ***

rem *** include Buttons***
#include "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Throw dice ***
dice = Random(0,9)
repeat
    rem *** Display prompt ***
    Print("Guess what my number is ")
    Sync()
    Sleep(2000)
    rem *** Get a value ***
    guess = GetButtonEntry()
    while guess < 0 or guess > 9
       Print("your guess must be between 0 and 9")
       Print("Enter your guess again(0 - 9) : ")
       Sync()
       Sleep(2000)
       guess = GetButtonEntry()
    endwhile
    rem *** Display message ***
    diff = dice - guess
    if diff > 2
        Print("You guess is too low")
    else if diff > 0
        Print("Your guess is slightly too low ")
    else if diff = 0
        Print("Correct")
    else if diff >= -2
        Print("Your guess is slightly too high")
    else
        Print("Your guess is too high")
    endif endif endif endif
until guess = dice

rem *** Display values ***
PrintC("My number was : ")
Print(dice)
PrintC("Your guess was : ")
```

```
Print(guess)
Sync()
do
loop
```

Activity 5.5

```
for j = 1 to 10
```

Activity 5.6

This code would display the values 1 to 10.

Activity 5.7

Modified code for *Tables* (12 times table):

```
rem *** 12 Times Table ***

rem *** Display title ***
Print("12 Times Table ")
Print("")
rem *** Display the table values ***
for c = 12 to 144 step 12
    Print(c)
next c
Sync()
do
loop
```

Activity 5.8

Modified version of *Tables*:

```
rem *** 12 Times Table ***

rem *** Display title ***
Print("12 Times Table ")
Print("")
for c = 144 to 12 step -12
    Print(c)
next c
Sync()
do
loop
```

Activity 5.9

Code for *OneTo*:

```
rem *** Display all values in a range ***

rem *** include Buttons functions ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()
rem *** Get limit ***
Print("Enter the upper limit")
Sync()
Sleep(2000)
num = GetButtonEntry()
rem *** Display numbers 1 to num ***
for c = 1 to num
    Print(c)
    Sync()
    Sleep(200)
next c
do
loop
```

Start value version of *OneTo*:

```
rem *** Display all values in a range ***

rem *** include Buttons functions ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()
rem *** Get lower limit ***
Print("Enter the lower limit")
Sync()
Sleep(2000)
start = GetButtonEntry()
rem *** Get upper limit ***
Print("Enter the upper limit")
Sync()
Sleep(2000)
num = GetButtonEntry()
rem *** Display numbers start to num ***
for c = start to num
    Print(c)
    Sync()
    Sleep(200)
next c
do
loop
```

Step size version of *OneTo*:

```
rem *** Display values in a range ***

rem *** include Buttons functions ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()
rem *** Get lower limit ***
Print("Enter the lower limit")
Sync()
Sleep(2000)
start = GetButtonEntry()

rem *** Get upper limit ***
Print("Enter the upper limit")
Sync()
Sleep(2000)
num = GetButtonEntry()
rem *** Get step size ***
Print("Enter the step size")
Sync()
Sleep(2000)
increment = GetButtonEntry()
rem *** Display numbers start to num ***
for c = start to num step increment
    Print(c)
    Sync()
    Sleep(200)
next c
do
loop
```

Activity 5.10

Code for *ForReal*:

```
rem *** Display values from 1 to 2 ***
for ch# = 1.0 to 2.0 step 0.1
    Print(ch#)
    Sync()
    Sleep(200)
next ch#
do
loop
```

Notice that the values displayed are 1.0 to 1.9.

Activity 5.11

Modified version of *ForReal*:

```
rem *** Display values from 1 to 2 ***
for ch# = 1.0 to 2.1 step 0.1
    Print(ch#)
    Sync()
    Sleep(200)
next ch#
do
loop
```

The display now runs from 1.0 to 2.0.

Activity 5.12

Code for *InTotal*:

```
rem *** Total input values ***

rem *** Include button functions ***
#include "Buttons.agc"
```

```
rem *** Set up buttons ***
SetUpButtons()
rem *** Set total to zero ***
total = 0
rem *** Read and sum 6 numbers ***
for c = 1 to 6
    Print("Enter number")
    Sync()
    Sleep(1000)
    no = GetButtonEntry()
    total = total + no
next c
PrintC("Total = ")
Print(total)
Sync()
do
loop
```

Activity 5.13

Code for *Shades*:

```
rem *** Display all shades of red ***
rem *** Set red intensity to ***
rem *** range from 0 to 255
for red = 0 to 255
    SetClearColor(red,0,0)
    Sync()
    Sleep(20)
next red
do
loop
```

Activity 5.14

Code for *Smallest*:

```
rem *** Find Smallest Number Entered ***

rem *** Include Button functions ***
#include  "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Get first number ***
Print("Enter number ")
Sync()
Sleep(2000)
no = GetButtonEntry()
rem *** Set smallest to first number ***
smallest = no
rem *** FOR 4 times DO ***
for c = 1 to 4
    rem *** Get next number ***
    Print("Enter number ")
    Sync()
    Sleep(1000)
    no = GetButtonEntry()
    rem *** If number smaller, record it ***
    if no < smallest
        smallest = no
    endif
next c
rem *** Display smallest value ***
PrintC("Smallest value entered was ")
Print(smallest)
Sync()
do
loop
```

Modified version of *Smallest*:

```
rem *** Find Largest Number Entered ***

rem *** Include Button functions ***
#include  "Buttons.agc"

rem *** Display buttons ***
SetUpButtons()
rem *** Get first number ***
Print("Enter number ")
Sync()
Sleep(2000)
no = GetButtonEntry()
rem *** Set largest to first number ***
largest = no
rem *** FOR 4 times DO ***
for c = 1 to 4
    rem *** Get next number ***
    Print("Enter number ")
    Sync()
    Sleep(1000)
    no = GetButtonEntry()
    rem *** If number larger, record it ***
    if no > largest
        largest = no
    endif
next c
rem *** Display largest value ***
PrintC("Largest value entered was ")
Print(largest)
Sync()
do
loop
```

Activity 5.15

Modified version of *SumDice*:

```
rem *** Total dice throws ***

rem *** set total to zero ***
total = 0
rem *** for 5 times do ***
for c = 1 to 5
   rem *** Display roll number ***
  PrintC("Roll number ")
  Print(c)
  Sync()
  Sleep(1000)
  rem *** throw both dice ***
  dice1 = Random(1,6)
  dice2 = Random(1,6)
  rem *** display throw number and dice values ***
  PrintC("dice 1 : ")
  PrintC(dice1)
  PrintC("          dice 2 : ")
  Print(dice2)
  Sync()
  Sleep(2000)
  rem *** if either dice is a 1 then quit loop ***
  if dice1 = 1 and dice2 = 1
    exit
  endif
  rem *** add dice throws to total ***
  total = total + dicel + dice2
next c
rem *** display final score ***
PrintC("Your final score was : ")
Print(total)
Sync()
do
loop
```

Activity 5.16

```
rem *** Display average of 6 scores ***

rem *** Include Button functions ***
#include "Buttons.agc"


rem *** Display buttons ***
SetUpButtons()
rem *** Set total to zero ***
total = 0
rem *** FOR 6 times DO ***
for c = 1 to 6
    rem *** Get valid score ***
    Print("Enter score ")
    Sync()
    Sleep(2000)
    score = GetButtonEntry()
    while score < 0 or score > 100
        Print("Score must lie between 0 and 100")
        Print("Enter score ")
        Sync()
        Sleep(2000)
        score = GetButtonEntry()
    endwhile
    rem *** Add score to total ***
    total = total + score
next c
rem *** Calculate average ***
average = total/6
rem *** Display average ***
```

```
  PrintC("Average = ")
  Print(average)
  Sync()
  do
  loop
```

Activity 5.17

No solution required.

Activity 5.18

The output would be:

```
-2  0
-2  1
-2  2
-2  3
-1  0
-1  1
-1  2
-1  3
 0  0
 0  1
 0  2
 0  3
 1  0
 1  1
 1  2
 1  3
```

On the computer screen, all output would occur on the same line with a slight delay between each set of values.

Activity 5.19

The code contains a `while` loop so we need to create three sets of test data to allow zero, one and more than one iteration of the loop.

Possible test values are:

	num	Expected Results (for *average*)
Test 1	0	0
Test 2	8,0	8
Test 3	12,6,0	9

Code for *Average*:

```
rem *** Calculate average of values entered ***


rem *** Include Button functions ***
#include "Buttons.agc"

rem *** Set up buttons ***
SetUpButtons()

total = 0
count = 0
Print("Enter number (0 to stop)")
Sync()
Sleep(2000)
num = GetButtonEntry()
while num <> 0
     total = total + num
     count = count + 1
     Print("Enter number (0 to stop)")
     Sync()
     Sleep(2000)
     num = GetButtonEntry()
endwhile
average = total / count
PrintC("Average is ")
Print(average)
Sync()
do
loop
```

When we run the program with the test data, it turns out that all the results are as we expected.

However, this is more by good fortune than the fact that the code is foolproof.

The line

```
average = total/count
```

would, in most languages, cause the program to crash when we did the first test. This is because *count* would have the value zero and hence the calculation would cause a division by zero error. However, as we saw back in Chapter 3, AGK BASIC returns zero when division by zero is performed - just the answer we want!

However, you really should guard against this problem. For example, if you were to rewrite your code in C++, then that division by zero calculation would cause a crash.

We can solve the problem by changing the code to

```
if count = 0
  average = 0
else
  average = total / count
endif
```

6

Resources - A First Look

In this Chapter:

- ❑ **Introducing Images**
- ❑ **Introducting Sprites**
- ❑ **Sound**
- ❑ **Music**
- ❑ **Introducing Text**
- ❑ **Introducing User Interaction**

Resources - A First Look

Introduction

Any additional visual components or files that we make use of within an AGK project are known as **resources**. Typical resources are: images, sounds, music, sprites, buttons and even text.

Every resource is assigned an integer ID value. No two resources of the same type may have the same ID. However, resources of different types may share the same ID. So, it's okay for an image, say, to have an ID of 1 and a sound resource to also have an ID of 1.

A resource's ID can be chosen by the programmer or automatically by the program itself.

Any separate files required by a resource must be copied into the project's *media* folder.

Images

Image Formats

The type of image you create using your camera or download from the web is a **bitmap** image. A bitmap image is constructed from a series of coloured dots known as **pixels**. You have probably come across this term before, since the resolution of any screen or camera is usually quoted in pixels. For example, the Apple iPad 1 & 2 screen has a resolution of 768 pixels by 1024 pixels.

The more pixels an image contains, the more detail it will hold. Therefore, we often talk about the resolution of an image as being its size in pixels. Many cameras can easily obtain image resolutions of over 4000 by 3000 pixels.

The other simple way to create a bitmap image is to use a paint package such as Adobe Photoshop or even the modest Paint program included with Microsoft Windows.

Many painting packages can resize images. This allows you to shrink or expand the number of pixels in an image. Decreasing the size of an image means that some of the details that were in the original image will be lost. On the other hand, increasing an image's size cannot create detail that was not there in the original and can often make the enlarged image look fuzzy and slightly out of focus.

Image files can be stored in many formats. Some formats will save an exact copy of the original image (known as **lossless** formats) but others lose a small amount of the original's detail (**lossy** formats). This second option doesn't sound like a great idea, but the reason such formats are popular - in fact, the most widely used of all - is because these **lossy** formats use compression techniques to create much smaller files. A lossy image can be stored in a file that is only 10% or even 5% of the **lossless** file equivalent.

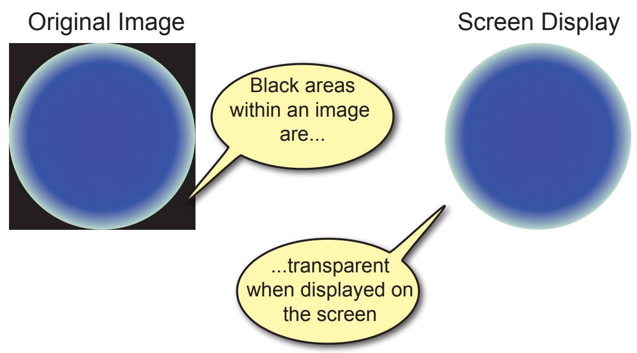
AGK BASIC recognises three image file formats. These are: BMP, PNG and JPG. BMP and PNG are lossless file formats and so should only be used for relatively small images; perhaps character figures and other visual components of a game. JPG is a lossy format and is ideal for use with photographs and larger graphics. The degree of compression used when saving a file in JPG format can be specified. Less compression means a better quality image but a larger file.

Image Transparency

Images are always rectangular in shape. So how do you create a game that displays a football or a spaceship or anything else that isn't rectangular? All we need to do is make part of the image transparent. In AGK, there are two methods of achieving transparent areas within a displayed image. One option is to make black areas within an image invisible on the screen (see FIG-6.1).

FIG-6.1

Black Pixel Transparency


Original Image
Screen Display
Black areas within an image are...
...transparent when displayed on the screen

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

➢ Only pixels which are truly black (red, green and blue intensities = 0) are made invisible. Part of the image which look black to you may not be completely black and therefore will not appear transparent when displayed.

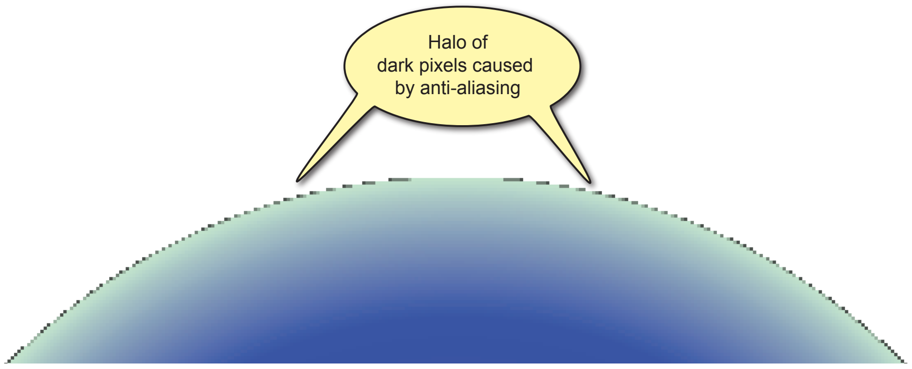
➢ You have to make sure that no part of the image that should remain visible contains black pixels.

➢ A final, and perhaps more subtle problem, is caused by anti-aliasing.

Anti-aliasing is an attempt by image manipulation software to blend the edges of objects within an image in such a way as to give a smooth transition from one object to the next. This helps hide the pixelated nature of a digital image and in most cases improves the image. However, it can cause havoc when trying to create a transparent background. When anti-aliasing has been used in an image, the transition from visible area to the black invisible area will have a halo of near-black pixels and this halo will be all too visible when your image appears on screen (see FIG-6.2).

FIG-6.2

Anti-aliasing



To avoid the halo problem, make sure anti-aliasing is switched off when you are creating an image. Using black pixels to produce transparency does have its limitations. For example, it does not allow us to create semi-transparent elements within an image.

JPG files cannot have an alpha channel.

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

We already know that an image is constructed from a sequence of pixels and that the colour of each pixel is determined by the intensity of its red, green and blue, components. These three colour components are sometimes referred to as the image's **colour channels**. Some image formats allow you to add a fourth channel known as the **alpha channel**. This channel is a grey-scaled overlay of the image surface and determines the transparency setting for every pixel within the image. In an area where the alpha channel is black, the image is fully transparent; where the alpha channel displays white, the image is opaque; and where the alpha channel is grey, the image is translucent. The shade of grey determines the degree of translucency.

FIG-6.3 shows an image, its alpha channel and how that image looks when displayed on screen.

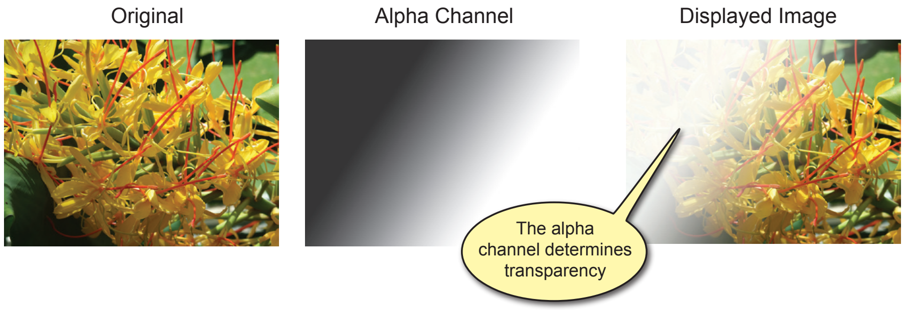


FIG-6.3

An Image with an Alpha Channel

The transparency is more obvious if we place a second image behind the original one (see FIG-6.4).

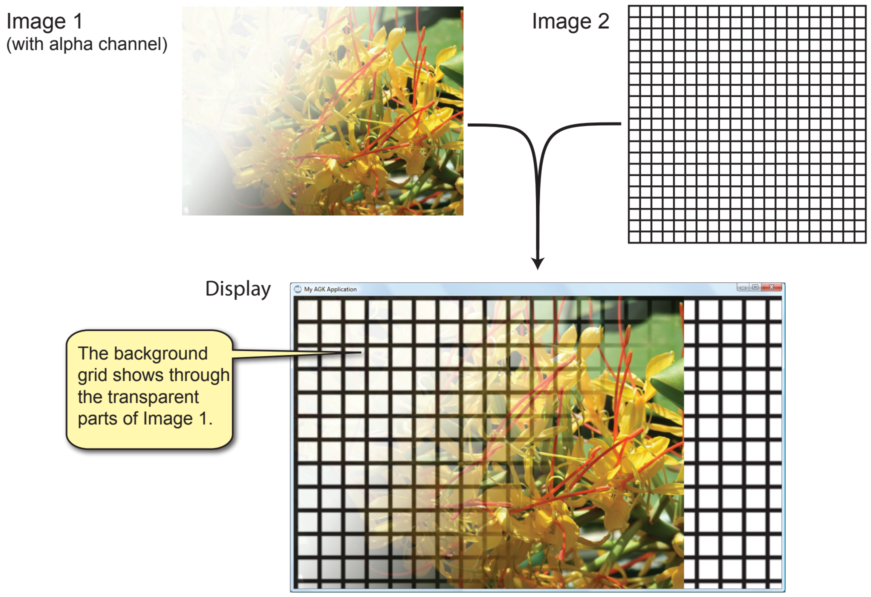


Image 1
(with alpha channel)
Image 2
Display
My AGK Application
The background grid shows through the transparent parts of Image 1.

FIG-6.4

Alpha Channel Transparency

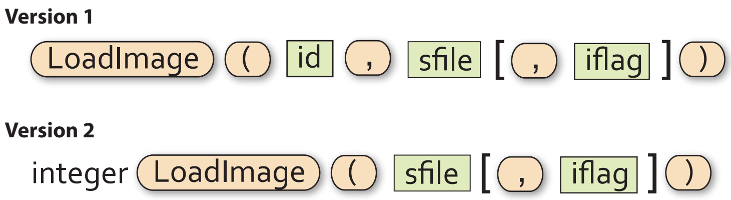
BMP and PNG files both allow alpha channel information to be stored (though in slightly different ways).

LoadImage()

If we want to display one or more images in a game, we need to start by copying the files containing the images into the AGK project's *media* folder. Next we need to issue a command to load each image into the game itself. This is done using the `LoadImage()` statement. There are two variations on this statement (see FIG-6.5).

FIG-6.5

LoadImage()


Version 1
LoadImage (id , sfile [, iflag])
Version 2
integer LoadImage (sfile [, iflag])

where:

id	is an integer value specifying the ID to be assigned to the image. This value must be 1 or above. No two images may have the same ID value.
sfile	is a string giving the name of the file containing the image. The file must be in the *media* folder for this project.
iflag	is an integer (0 or 1) which is used to determine how transparency is handled when the image is displayed. If *iflag* has the value zero, then the alpha channel of the image sets the transparency; if the value is 1, then the alpha channel is ignored and all black pixels within the image are made invisible. A value of zero is assumed if this parameter is omitted.

Using the first version of this command, you need to specify the ID being assigned to the image for the duration of the program. For example, if the first image to be loaded is called "*ball.bmp*", then we would load the image using the statement

```
LoadImage(1,"ball.bmp",1)
```

This will assign the ID value of 1 to the image and black pixels will be invisible. Alternatively, we could use version 2 of the statement and write

```
id = LoadImage("ball.bmp",1)
```

This time the program decides on the ID to be assigned, but IDs are assigned in ascending order starting at 10001, so, as long as this is the first image to be loaded it will be assigned an ID of 10001.

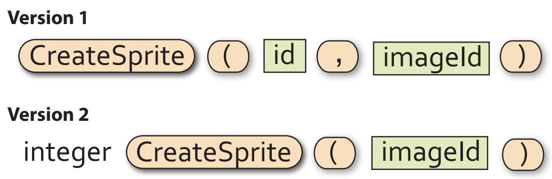
Using the second version guarantees that we will not attempt to assign the same ID to two different images (which would, in any case, produce an error).

CreateSprite()

Although all images need to be loaded before they can be used, in order to see an image on the screen, you'll need to load that image into a sprite. To do this you need to create a sprite and specify the image to be displayed by the sprite. This is done using the `CreateSprite()` statement (see FIG-6.6).

FIG-6.6

CreateSprite()


Version 1
CreateSprite (id , imageId)
Version 2
integer CreateSprite (imageId)

where:

id	is an integer value specifying the ID to be assigned to the sprite. This value must be 1 or above. No two sprites may have the same ID value.
imageId	is an integer value specifying the ID of the image being copied into the sprite. This image must previously have been loaded using a `LoadImage()` statement. Use 0 to create a white sprite without an image.

At this stage we can think of a sprite as nothing more than an image which appears on the screen. But, as we will discover later, there are many sprite-related commands which allow us to do various operations such as move, rotate, resize and detect sprite collisions.

Like the two versions of `LoadImage()`, the two options in the `CreateSprite()` statement allow you to choose between deciding on the ID number yourself (version 1) or letting the program decide for you (version 2 - assigned values start at 10001).

In the example we are about to create, we will assign our own ID numbers since it uses only a single image and a single sprite. So, to create a sprite showing the ball image, we would first load the image and then create the sprite:

```
LoadImage(1,"ball.bmp",1)
CreateSprite(1,1)
```

Notice that the image and sprite have both been assigned an ID of 1. This is not a problem since they are two different types of objects (image and sprite). Only when you assign the same ID to two objects of the same type do you cause an error. Now we are ready to create a program to display our first image (see FIG-6.7).

FIG-6.7

Displaying a Sprite

```
rem *** First Sprite ***
rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
Sync()
do
loop
```

When a sprite is first created, its top left corner is at position (0,0) - the top left corner of the app window.

Activity 6.1

Create a new project called *FirstSprite*. Compile the default code in order to create the project's *media* folder. From the files you downloaded to accompany this book, go to the *AGKDownloads/Chapter 6* folder and copy the file *ball.bmp* to the project's *media* folder.

Change the contents of *main.agc* to match that given in FIG-6.7. Run and save the project. What is strange about the image?

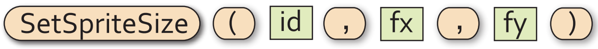
AGK has a problem with sizing the image. Since we are working with a percentage-based screen layout, it has no idea exactly how large to make the sprite. It handles this by assuming the physical size of the image represents the percentage required. The ball image is 64 pixels wide by 64 pixels high, so AGK assumes you want the image to take up 64% of the width and 64% of the height of the app window. Unfortunately, this is nowhere near the actual size we want.

SetSpriteSize()

The `SetSpriteSize()` statement allows use to specify the dimensions of a sprite. The sizes are given as a percentage of the screen, or in virtual pixels, depending on the option chosen when the program was created. The statement has the format shown in FIG-6.8.

FIG-6.8

SetSpriteSize()



where:

id	is the integer value previously assigned as the ID of the sprite to be resized.
fx	is a real value giving the width required. This value is given as a percentage of the screen width or in virtual pixels as appropriate.
fy	is a real value giving the height required (percentage or virtual pixels).

So, if we wanted the ball sprite to occupy only 10% of the screen, we would use the line:

```
SetSpriteSize(1,10,10)
```

Activity 6.2

Modify *FirstSprite* by adding the `SetSpriteSize()` statement given above. Run the program and see how this changes the image displayed.

Change the height setting in *setup.agc* to 1024. Rerun the program. How is the sprite affected? Save your project.

As you can see from Activity 6.2, making the sprite 10% in both directions works only when the app window is square. Increasing the app window height also means an increase in the height of the sprite and our ball is no longer circular.

To solve this problem, `SetSpriteSize()` allows you to set the actual size of one dimension and use the value -1 for the other. When you choose this option, AGK works out the second dimension automatically to ensure that the sprite retains its original shape. For example, if we set the *fx* parameter to 10 and *fy* to -1 using the line

```
SetSpriteSize(1,10,-1)
```

the sprite will return to its round shape.

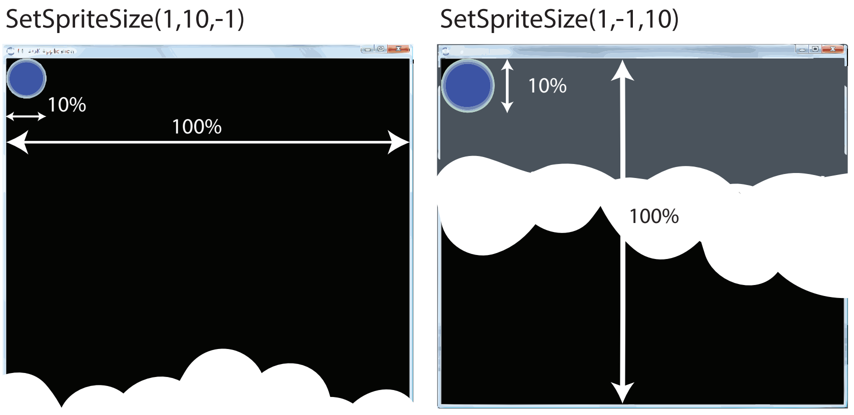
Of course, setting the *fy* to 10 and *fx* to -1 with

```
SetSpriteSize(1,-1,10)
```

will still result in a round ball, but it will be larger since 10% of the app window's height is much greater than 10% of its width (see FIG-6.9).

FIG-6.9

How Sprite Size Changes



Activity 6.3

Modify *FirstSprite* to use the -1 parameter in `SetSpriteSize()`. Try out both options, making the width -1 on the first run and the height -1 on the second run.

Save your project.

The only problem now with our sprite app is that, since the app window background is black, we really can't see if the black areas of the sprite are, indeed, invisible.

Activity 6.4

Add a `SetClearColor()` statement to your *FirstSprite* program to create a white background. (You'll also need to add an extra `Sync()` statement.)

Are the black pixels within the ball image invisible?

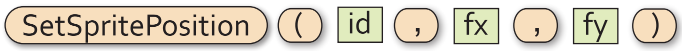
Save your project.

SetSpritePosition()

An existing sprite can be moved to a new position on the screen using the `SetSpritePosition()` statement which has the format shown in FIG-6.10.

FIG-6.10

SetSpritePosition()



where:

id	is the integer value previously assigned as the ID of the sprite to be moved.
fx	is a real value giving the new x-coordinate (percentage or virtual pixels).
fy	is a real value giving the new y-coordinate. Measured in virtual pixels or percentage.

By default, it is the top left corner of a sprite that is placed at the position specified.

Activity 6.5

In *FirstSprite*, add a two second delay and then move the sprite to the centre of the app window. Test and save your project.

By placing the `SetSpritePosition()` statement within a `for` loop and using the loop counter as a parameter, we can get the sprite to travel across the window.

Activity 6.6

Remove your last modification from *FirstSprite* and replace it with the following code:

```
for p = 1 to 100
      SetSpritePosition(1,p,p)
      Sync()
      Sleep(50)
next p
```

Test the new version of the project.

SetSpriteDepth()

The program in FIG-6.11 is an extension of your *FirstSprite* project and demonstrates one sprite passing "behind" another.

FIG-6.11

Demonstrating Sprite Depth

```
rem *** Sprite Depth ***
rem *** Clear screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load images ***
LoadImage(1,"ball.bmp",1)
LoadImage(2,"poppy.bmp",1)
rem *** Create ball sprite ***
CreateSprite(1,1)
SetSpriteSize(1,10,-1)
rem *** Create poppy sprite ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
    Sleep(50)
next p
do
loop
```

Activity 6.7

Modify your *FirstSprite* project to match the code given in FIG-6.11.

Test and save your project.

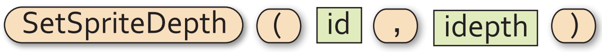
The ball passes "behind" the poppy because the ball sprite was created before the poppy. If we had wanted the ball to pass over the poppy, then we could have achieved this by having created the ball sprite after the poppy sprite. But another option is available; we can adjust the depth of a sprite using the `SetSpriteDepth()` statement. Sprite depth can be set to any value from 0 to 10000.

In original hand-drawn cartoons, the overall image is made up of a layer of transparent acetates. Different elements of the picture were drawn on different acetates. Those elements on the top-most acetate were at the "front" and those on the bottom acetate were at the "back". AGK depth settings are equivalent to those acetate layers: depth 0 is at the "front"; depth 10000 is at the "back".

The format of the `SetSpriteDepth()` statement is shown in FIG-6.12.

FIG-6.12

SetSpriteDepth()



where:

id	is the integer value previously assigned as the ID of the sprite.
idepth	is an integer value giving the layer setting. A lower number will bring the sprite "forward" towards the top layer. This value can be in the range 0 to 10,000.

When a sprite is created, it is assigned a default layer of 10. Sprites on the same layer have a depth determined by the order in which they were created (as we have already seen).

Activity 6.8

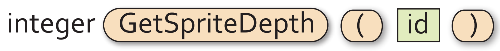
Modify *FirstSprite*, assigning the ball sprite to layer 9 immediately after its creation. How does this affect the program's display? Save your project.

GetSpriteDepth()

To determine the current depth of a sprite, use the `GetSpriteDepth()` statement (see FIG-6.13).

FIG-6.13

GetSpriteDepth()



where:

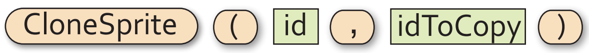
id	is the integer value previously assigned as the ID of the sprite.

CloneSprite()

You can make a copy of a sprite using the `CloneSprite()` statement. This will make an exact copy of the sprite specified. The statement's format is shown in FIG-6.14.

FIG-6.14

CloneSprite()



where:

id	is the integer value of the ID to be assigned to the new sprite.

idToCopy is an integer value giving the ID of the existing sprite to be cloned.

Whatever characteristics have been set for the original sprite (size, transparency, depth, etc.) will be duplicated in the clone.

Activity 6.9

Modify *FirstSprite*, making a copy of the poppy sprite and positioning it at (20,20).

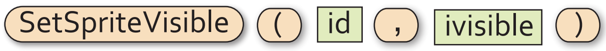
Assign the new sprite a depth setting of 8. What happens as the ball passes the two poppies? Save your project.

SetSpriteVisible()

We can make a sprite invisible - and make it reappear - using the `SetSpriteVisible()` statement which has the format shown in FIG-6.15.

FIG-6.15

SetSpriteVisible()



where:

id is the integer value previously assigned as the ID of the sprite.

ivisible is an integer value (0 or 1) specifying that the sprite is to be hidden (0) or made visible (1).

Activity 6.10

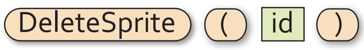
Modify *FirstSprite* so that the two poppy sprites are hidden after the ball has moved to the bottom of the screen. Save your project.

DeleteSprite()

When a sprite is no longer required by a program, that sprite can be deleted. Although deletion is not necessary, it does free up resources on the machine which can, in turn, speed up your game. Sprites are deleted using the `DeleteSprite()` statement whose format is shown in FIG-6.16.

FIG-6.16

DeleteSprite()



where:

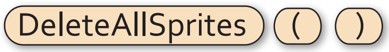
id is an integer value giving the ID of the sprite to be deleted.

DeleteAllSprites()

If your program contains several sprites, they can all be deleted, using the `DeleteAllSprites()` statement (see FIG-6.17).

FIG-6.17

DeleteAllSprites()

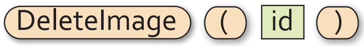


DeleteImage()

When an image is no longer required by a sprite, or when the sprite using an image has been deleted, then that image can be deleted, thereby freeing up further resources. To delete an image we use the `DeleteImage()` statement (see FIG-6.18).

FIG-6.18

DeleteImage()



where:

id	is an integer value giving the ID of the image to be deleted.

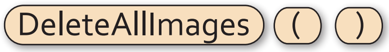
Deleting a resource only deletes it from the computer's memory; the actual file containing the resource is not affected.

DeleteAllImages()

Rather than delete images individually, you can delete every loaded image using the `DeleteAllImages()` statement (see FIG-6.19).

FIG-6.19

DeleteAllImages()



Of course, you should only call this statement when every image in the program is no longer being used by other program elements such as a sprite.

There are many more sprite commands and these will be covered in later chapters.

Sound

Sound files, like image files, come in many different formats. And like those for images, some formats are lossy, but have small file sizes, while others are lossless with larger file sizes. The current version of AGK will handle only uncompressed WAV sound files.

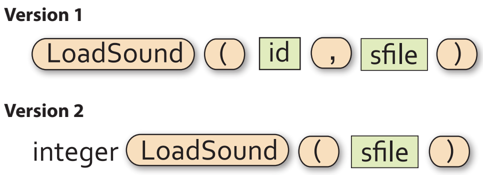
To play a sound, the file containing that sound must first be copied into the project's *media* folder. Within the program we can then load and play the file.

LoadSound()

Like images, sounds must be loaded before they can be used. This is done using the `LoadSound()` statement (see FIG-6.20).

FIG-6.20

LoadSound()



where:

id	is an integer value specifying the ID to be assigned to the sound file.
sfile	is a string giving the name of the file to be loaded. This must be a WAV file and must be stored in the project's *media* folder.

Automatically assigned ID values start at 1.

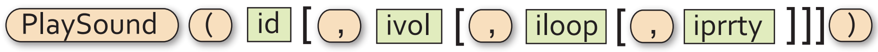
In the first version of the statement the program chooses the ID number; in the second version the ID value is automatically selected by AGK and returned by the statement.

PlaySound()

Once loaded, a sound file can be played using the `PlaySound()` statement (see FIG-6.21).

FIG-6.21

PlaySound()



where:

id	is an integer value specifying the ID previously assigned to the sound.
ivol	is an integer value (0 to 100) representing the volume setting. The default setting is 100.
iloop	is an integer value (0 or 1) which determines if the sound is to play continuously. If set to 0, the sound will play only once; if set to 1, the sound will be repeated. Zero is the default value.
iprrty	is an integer value which is designed to be used to set the sound's priority. This option is currently not implemented.

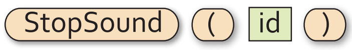
Several sound files can be played at the same time.

StopSound()

When a sound is set to play only once, it will, obviously, stop when the end of the file is reached, but if you want playing to stop prematurely, you can do so using the `StopSound()` statement. This statement has the format shown in FIG-6.22.

FIG-6.22

StopSound()



where:

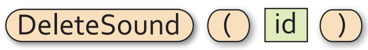
id	is an integer value giving the ID of the sound that is to be stopped.

DeleteSound()

When a sound resource is no longer required, it is best to delete that resource from your program. This can be done using the `DeleteSound()` statement (see FIG-6.23 for format).

FIG-6.23

DeleteSound()



where:

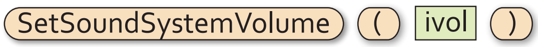
id	is an integer value giving the ID of the sound that is to be deleted.

SetSoundSystemVolume()

Although the volume of a specific sound is set when that sound is first loaded and cannot be adjusted later, the system volume can be adjusted at any time using the `SetSoundSystemVolume()` statement which has the format shown in FIG-6.24.

FIG-6.24

SetSoundSystemVolume()



where:

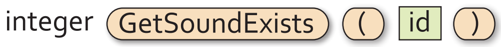
ivol	is an integer (0 to 100) giving the percentage volume adjustment. For example, 50 would give half volume, 100 would leave the volume unchanged.

GetSoundExists()

You can check that a sound with a specific ID value currently exists using `GetSoundExists()` (see FIG-6.25).

FIG-6.25

GetSoundExists()



where:

id	is an integer value giving the ID of the sound to be checked.

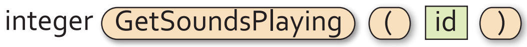
The statement will return 1 if a sound of the specified ID currently exists; otherwise zero is returned.

GetSoundsPlaying()

We can also check the number of instances of a sound that are playing at the same time. `GetSoundsPlaying()` returns the number of instances of a specified sound currently in existence (see FIG-6.26).

FIG-6.26

GetSoundsPlaying()



where:

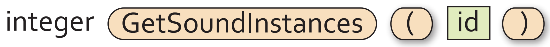
id	is an integer value giving the ID of the sound whose number of instances is to be returned.

GetSoundInstances()

The `GetSoundsInstances()` statement performs exactly the same purpose as `GetSoundsPlaying()` and so the two statements are interchangeable. The statement's syntax is shown in FIG-6.27.

FIG-6.27

GetSoundInstances()



where:

id	is an integer value giving the ID of the sound whose number of instances is to be returned.

Activity 6.11

Start a new project called *Sounds*. Compile the default code to create the *media* folder. Copy the file *J1to10.wav* from the *AGKDownloads/Chapter6* to the project's *media* folder.

Activity 6.11 (continued)

Recode the contents of *main.agc* to read:

```
LoadSound(1,"J1to10.wav")
PlaySound(1)
do
loop
```

Make sure the sound is activated and the volume turned up on your computer.

Compile and run the program. Does the sound play? Save your project.

When the program plays a sound file it does not halt execution of the other statements in your program while the sound is played. It merely passes the sound file details to your sound card, leaves the sound card to deal with playing the file, and then gets on with executing the other statements in your program.

Activity 6.12

Modify the code in *Sounds* so that it displays the numbers 1 to 10 as the sound file plays. The code for this is:

```
LoadSound(1,"J1to10.wav")
PlaySound(1)
for c = 1 to 10
      Print(c)
      Sync()
      Sleep(1000)
next c
do
loop
```

Test the program. Does the sound stop when the `Sleep(1000)` statement is executed? Save your project.

We have seen in previous chapters that the `Sleep()` statement halts the program for a specified time. However, since the sound file is being handled by the sound card, any sounds already being played are not affected by the `Sleep()` statement.

Activity 6.13

In this Activity we are going to examine what is required in order to have a sound file played repeatedly.

Remove the `for..next` loop and its loop body from *Sounds*.
Change the line

```
PlaySound(1)
```

to

```
PlaySound(1,100,1)
```

so that the sound should play repeatedly at full volume. Run the program. Does the sound play more than once?

Inside the `do..loop` add the line

```
Sync()
```

How does this affect the playing of the sound file? Save your project.

So the `Sync()` statement needs to be executed in order for the sound to play continuously. This is because the `Sync()` statement does more than just update the screen. It handles details about other things within the program including making sure sound files are replayed when appropriate.

Music

Music files are handled separately from sound files and although some of the commands for handling music look very similar to those for sounds, there are major differences.

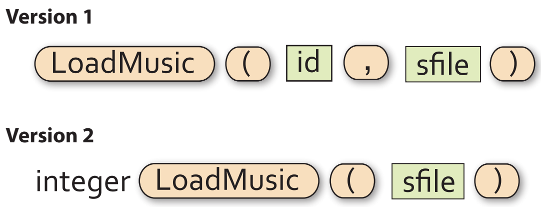
AGK currently plays only MP3, OGG Vorbis and ACC formatted music files.

LoadMusic()

The `LoadMusic()` statement loads a specified music file and assigns it an ID number. The statement has the format shown in FIG-6.28.

FIG-6.28

LoadMusic()



where:

id is an integer value specifying the ID to be assigned to the music file.

sfile is a string giving the name of the file to be loaded, This must be an MP3, OGG Vorbis or AAC file and must be stored in the project's *media* folder.

Automatically assigned ID values start at 1.

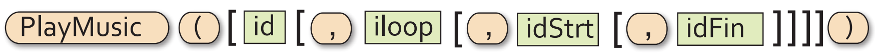
In the first version of the statement, the programmer chooses the ID number; in the second version, the ID value is automatically selected by AGK and returned by the statement.

PlayMusic()

Once loaded, a music file is played using the `PlayMusic()` statement (see FIG-6.29).

FIG-6.29

PlayMusic()



where:

id is an integer value giving the ID of the music file to be played.

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

iloop is an integer value (0 or 1) which determines if the music is to play continuously. If set to 0, the music will play only once; if set to 1, the music will be repeated. Zero is the default value.

idStrt is an integer value giving the lowest ID of the list of music files to be played.

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

This command will play all or most of the MP3 files stored in the *media* folder without explicitly specifying all the ID numbers. To stop this you need to use the longest form of the command and state explicitly which file or group of files are to be played.

The simplest version of this command is

```
PlayMusic()
```

which will play the music file with the lowest ID. For example, if a program started with the lines

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

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

and followed this with

```
PlayMusic()
```

then *TrackA* would be played first and then all other tracks played in sequence.

```
PlayMusic(2,0)
```

would play *TrackB* followed by *TrackC*, *TrackD* and *TrackE*. The tracks would be played once only.

```
PlayMusic(3,1)
```

would play *TrackC, TrackD,* and *TrackE* and then play all five tracks continuously.

```
PlayMusic(1,1,3,5)
```

would play *TrackA, TrackB* then repeat *TrackC*, *TrackD* and *TrackE* continuously.

```
PlayMusic(3,0,3,3)
```

would play *TrackC* once only.

Using this command also requires you to add a `Sync()` statement within the `do..loop` structure.

Activity 6.14

For copyright reasons, no MP3 files are included in the downloads for this book.

Start a new project called *Music*. Compile the default code to create the project's *media* folder. Copy three of your own MP3 files into the *media* folder.

Modify *main.agc* to load all three files but play only the last one. The file should be played only once. Test and save your code.

PauseMusic()

You can pause a playing MP3 file using the `PauseMusic()` statement. This has the format shown in FIG-6.30.

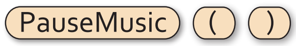


FIG-6.30

PauseMusic()

Note that there is no need for an ID parameter since only one music file can be playing at any instant.

ResumeMusic()

A paused MP3 file can be resumed from the point where it paused using the `ResumeMusic()` statement (see FIG-6.31).

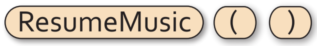


FIG-6.31

ResumeMusic()

StopMusic()

To stop a music file completely use `StopMusic()` (see FIG-6.32).

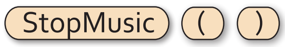


FIG-6.32

StopMusic()

DeleteMusic()

When a music resource is no longer required you can use the `DeleteMusic()` statement to free up the memory occupied by the file (see FIG-6.33).

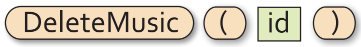


FIG-6.33

DeleteMusic()

where:

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

We can determine various characteristics about music files from several other music statements.

GetMusicExists()

The `GetMusicExists()` statement returns 1 if a music resource of a specified ID currently exists; otherwise zero is returned (see FIG-6.34).

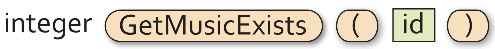


FIG-6.34

GetMusicExists()

where:

id	is an integer value giving the ID of the music resource to be checked.

SetMusicFileVolume()

You can set the volume of a specific music file using the `SetMusicFileVolume()` (see FIG-6.35).

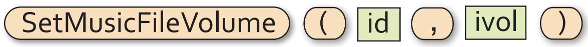


FIG-6.35

SetMusicFileVolume()

where:

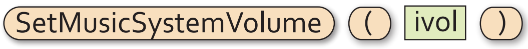
id	is an integer value giving the ID of the music whose volume is to be changed.
ivol	is an integer giving the volume as a percentage of full volume (0 - silent; 100 - full volume).

SetMusicSystemVolume()

To set the volume for every music track, the `SetMusicSystemVolume()` statement can be used (see FIG-6.36).

FIG-6.36

SetMusicSystemVolume()



where:

ivol	is an integer giving the volume as a percentage of full volume (0 - silent; 100 - full volume).

Detecting User Interaction

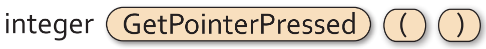
Most programs react to the user clicking a mouse or touching a pressure-sensitive screen. AGK uses three main commands to detect a mouse/screen press.

GetPointerPressed()

One of these commands is the `GetPointerPressed()` statement which has the format shown in FIG-6.37.

FIG-6.37

GetPointerPressed()



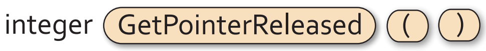
The statement returns 1 immediately the press occurs. Before and after that instant, zero is returned.

GetPointerReleased()

A complementary statement is `GetPointerReleased()` which returns 1 the instant the mouse button is released, or the finger lifted from the screen. This statement has the format shown in FIG-6.38.

FIG-6.38

GetPointerReleased()

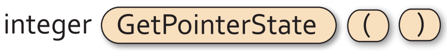


GetPointerState()

This third statement returns 1 while the button or finger is being pressed down and returns 0 when the button/finger is not pressed. Note this is different from the first two statements which only return 1 for a single instant as the mouse/finger is pressed/lifted. The `GetPointerState()` command has the format shown in FIG-6.39.

FIG-6.39

GetPointerState()



The code in FIG-6.40 demonstrates the use of the `GetPointerPressed()` and `GetPointerReleased()` statements.

FIG-6.40

Using Pointer Statements

```
Sync()
do
    rem *** Check for press ***
    if GetPointerPressed()=1
        Print("Pressed")
    endif
    rem *** Check for release ***
    if GetPointerReleased()=1
        Print("Released")
    endif
    Sync()
loop
```

Notice that for the first time, the main code is within the `do..loop` structure which loops continually while testing for the button/screen press.

Activity 6.15

Start a new project called, *PressedFlower* and change the code in *main.agc* to match that given in FIG-6.40.

Test the program and check that you can see messages as you press and release the mouse button. Save your project.

If we are not interested in detecting the exact moment the button is pressed or released, but want to know if the button/finger is currently pressed down/touching the screen or up/not touching the screen, then the `GetPointerState()` command will be more useful.

Activity 6.16

Modify the code in *PressedFlower* to read:

```
Sync()
do
    if GetPointerState()=1
        Print("Pressed")
    else
        Print("Released")
    endif
    Sync()
loop
```

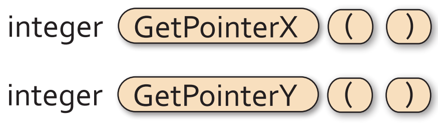
Test the new code. How do the messages that appear on the screen differ from those displayed by the previous version of the program? Save your project.

GetPointerX() and GetPointerY()

We can find out the exact position on the screen where a press has occurred using `GetPointerX()` (which returns the x-coordinate) and `GetPointerY()` (which returns the y-coordinate). The formats for these two statements are shown in FIG-6.41.

FIG-6.41

GetPointerX()
GetPointerY()



Activity 6.17

Modify the code in *PressedFlower* by removing the line

```
        Print("Pressed")
```

and replacing it with

```
        PrintC(GetPointerX())
        PrintC(" ")
        Print(GetPointerY())
```

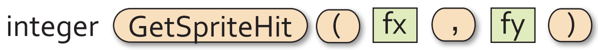
Test and save your project.

GetSpriteHit()

We can find out if a particular screen position is over a sprite using the `GetSpriteHit()` command. This is useful for finding out if the user has, for example, clicked/pressed on a sprite. The command's format is shown in FIG-6.42.

FIG-6.42

GetSpriteHit()



where:

fx, fy are real numbers giving the position within the app window to be tested. The values will represent percentages or virtual coordinates depending on the window setup.

If the location is over a sprite, the sprite ID is returned, otherwise zero is returned.

Activity 6.18

Modify *PressedFlower* by removing all of the code within the `do..loop` structure.

Add code to display a sprite showing *poppy.bmp* at the centre of the app window (set the sprite's width to 15%).

To hide the poppy when it is clicked on, change the code within the `do..loop` structure to:

```
        if GetPointerPressed()=1
              x# = GetPointerX()
              y# = GetPointerY()
              hit = GetSpriteHit(x#,y#)
              if hit <> 0
                    SetSpriteVisible(1,0)
              endif
        endif
        Sync()
```

Test and save your project.

Text Resources

We've already seen how to display information on the screen using the `Print()` statement, but its main limitation is that we cannot choose the exact position at which the output is to appear. This will be a critical requirement for any game.

Luckily, AGK offers a second and more controlled way of creating textual output; **text resources**. Just like images, sprites, sound, and music resources, text resources are created and assigned a unique ID.

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

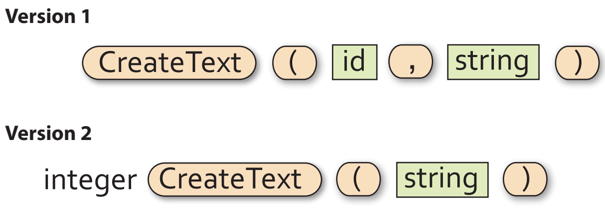
A few of the many statements available for manipulating text resources are described here.

CreateText()

The `CreateText()` statement allows us to create a new text resource. The statement has the format shown in FIG-6.43.

FIG-6.43

CreateText()



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:

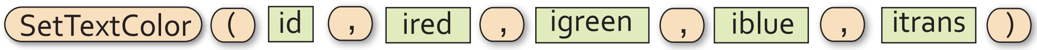
```
CreateText(1, "Hello world")
```

SetTextColor()

We can select the color and transparency of the text using the `SetTextColor()` statement (see FIG-6.44).

FIG-6.44

SetTextColor()



where:

The default colour for a text resource is white.

id	is an integer value specifying the ID of the text resource whose colour is to be set.
ired	is an integer value specifying the intensity of the red component of the colour. Range 0 to 255.
igreen	is an integer value specifying the intensity of the green component of the colour. Range 0 to 255.
iblue	is an integer value specifying the intensity of the blue component of the colour. Range 0 to 255.

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

For example, if we have already created a text resource with an ID of 1, then we can display that text in opaque black using the line:

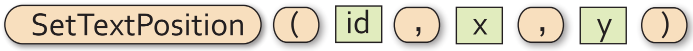
```
SetTextColor(1,0,0,0,255)
```

SetTextPosition()

By default, text will appear in the top left corner of the app window. To position it elsewhere we need to use the `SetTextPosition()` statement which has the format shown in FIG-6.45).

FIG-6.45

SetTextPosition()



where:

id is the integer value previously assigned as the ID of the text to be moved.

x is a real value giving the new x-coordinate. This will be in virtual pixels or percentage depending on the coordinate system defined when the app window was created.

y is a real value giving the new y-coordinate measured in virtual pixels or percentage.

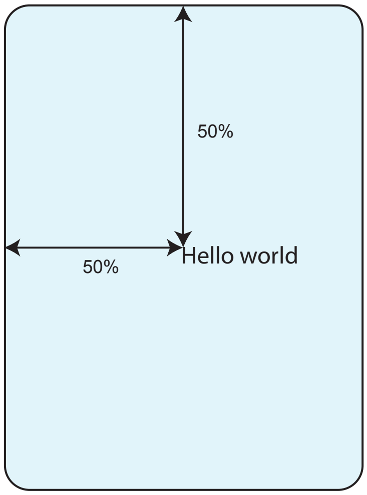
We could place text resource 1 at the centre of the app window using the statement:

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

The position (50,50) refers to the top left part of the text (see FIG-6.46).

FIG-6.46

Positioning a Text Resource

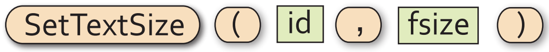


SetTextSize()

The size of the text can be adjusted using the `SetTextSize()` statement (see FIG-6.47).

FIG-6.47

SetTextSize()



where:

id	is the integer value previously assigned as the ID of the text to be resized.
fsize	is a real value specifying the height of the characters within the text. This is measured in percentage or virtual pixels depending on the setup. The width is calculated automatically.

The default size for all text output is 4. Remember also that the larger the text becomes, the more obvious the limitations of the images from which it is derived.

We could change the size of the text displayed by text resource 1 from the default 4 to 6 using the statement:

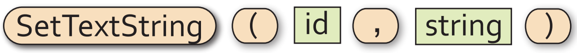
```
SetTextSize(1,6)
```

SetTextString()

The actual text contained within a text resource can be changed using the `SetTextString()` statement (see FIG-6.48).

FIG-6.48

SetTextString()



where:

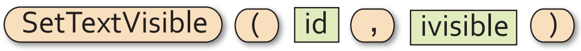
id	is the integer value previously assigned as the ID of the text resource whose text is to be changed.
string	is the new string to be assigned to the text resource.

SetTextVisible()

You can hide a text resource or make it reappear using the `SetTextVisible()` statement (see FIG-6.49).

FIG-6.49

SetTextVisible()



where:

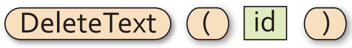
id	is the integer value previously assigned as the ID of the text resource to be operated on.
ivisible	is an integer value (0 or 1) used to hide or display the text. (0 - hide ; 1 - show)

DeleteText()

When a text resource is no longer required, it should be deleted, thereby freeing up memory resources. This is done using the `DeleteText()` statement (see FIG-6.50).

FIG-6.50

DeleteText()



where:

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

DeleteAllText()

If your program contains several text resources and you wish to remove all of them, use `DeleteAllText()` (see FIG-6.51).

FIG-6.51

DeleteAllText()

Using a Text Resource

The program below demonstrates most of the text resource statements we have covered here. The purpose of the code is to display a sequence of dots. Starting with one dot and increasing to 10 before starting again at one dot. This sequence is repeated five times before the program stops. A simple animation such as this might be used to indicate to the user that the program is busy.

The program's logic can be described in structured English as:

```
Create empty text resource
Set text colour
Set text size
Set text position
FOR 5 times DO
    Create empty string
    FOR dots = 1 TO 10 DO
        Add dot to string
        Place string in text resource
        Wait 200 msecs
    ENDFOR
    Empty text resource
    Wait 1 sec
ENDFOR
Delete text resource
```

The code for the program is shown in FIG-6.52.

FIG-6.52

Using a Text Resource

```
rem *** Text Resource demo ***

rem *** Create empty string ***
CreateText(1,"")
rem *** Set resource attributes ***
SetTextPosition(1,15,30)
SetTextColor(1,250,250,0,255)
SetTextSize(1,10)
rem *** FOR 5 times DO ***
for c = 1 to 5
    rem *** Empty string ***
    text$ = ""
    for dots = 1 to 10
        rem *** Add dot to string ***
        text$ = text$+"."
        rem *** Place string in text resource ***
        SetTextString(1,text$)
        Sync()
        rem *** Wait 200 msecs ***
        Sleep(200)
    next dots
```

FIG-6.52
(continued)

Using a Text Resource

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

Activity 6.19

Start a new project called *UsingText* and modify the code in *main.agc* to match that given in FIG-6.52. Test the program.

Modify the code to use the underscore character (_) instead of the full stop.

Test and save your project.

Later

This chapter has covered all of the statements available for manipulating sound and music resources. However, there are many other commands that can be used with images, sprites, text and user input which are not covered here. These will be explained in later chapters.

Summary

- ➢ Resources is the name given to other elements added to a project. These can be images, sounds, music, sprites, virtual buttons, or text.
- ➢ A resource needs to be created and assigned an ID before it can be used.
- ➢ No two resources of the same type may be assigned the same ID number.
- ➢ Resources of different types may have identical ID numbers.
- ➢ As a general rule, resources should be deleted when no longer required.
- ➢ Files containing resources must be stored in the project's *media* folder.
- ➢ Most images are constructed from colour dots known as pixels.
- ➢ An image constructed from pixels is known as a bitmap image.
- ➢ Bitmap images can be stored in many different formats.
- ➢ Lossless formats save an exact copy of an image but create large files.
- ➢ Lossy formats save a degraded copy of the image but create smaller files.
- ➢ AGK can handle three bitmap formats: BMP, PNG, and JPG.
- ➢ BMP and PNG are lossless file formats; JPG is a lossy file format.
- ➢ Images can contain transparent elements.
- ➢ Transparency can be achieved in one of two ways: by making all black pixels

invisible or by adding an alpha channel to the image.

- Alpha channels allow degrees of translucency.
- When creating an image in which black elements are to be made invisible make sure that the image has not been created using anti-aliasing.
- Anti-aliasing can cause problems around the edges of objects within an image.
- Images need to be loaded into AGK and given a unique ID number.
- To display an image on the screen it must first be loaded into a sprite.
- Using the default setup, screen distances are given in percentage terms and sprites use the pixel size of the image it contains as a percentage value when determining the size of the image.
- Sprites can be resized, moved, and made invisible.
- Sprites can be placed on different layers.
- There are 10,001 layers numbered 0 to 10,000.
- Layer 0 is the top layer; layer 10,000 is the bottom layer.
- A sprite placed on a higher layer will pass in front of a sprite placed on a lower layer.
- A sprite can be cloned.
- A sprite can be made invisible.
- Deleting a sprite frees up the resources it requires.
- Sound files must be in uncompressed WAV format.
- A sound can be set to play one time only or repeatedly.
- The volume of an individual sound can be set only when playing starts.
- The overall system volume can be modified at any time.
- Music files must be in MP3 OGG Vorbis or AAC formats.
- By default, all music files are played once when a `PlayMusic()` command is issued.
- Basic user interaction allows us to detect a screen touch or mouse button press.
- It is possible to detect when:

  the mouse button/screen is first pressed
  the mouse button/screen is first released
  the current state of the mouse button/screen - pressed or unpressed.
- We can detect if a mouse/screen press occurs over a sprite.
- Using a text resource allows us to control attributes of a string.
- The string within a text resource can be modified, resized, positioned, coloured, and made transparent.

Solutions

Activity 6.1

Although the image is only 64 x 64 pixels it appears much larger within the app window.

Activity 6.2

Modified *FirstSprite*:

```
rem *** First Sprite ***

rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Resize sprite ***
SetSpriteSize(1,10,10)
Sync()
do
loop
```

The sprite now occupies 10% of the width and height of the app window. Because the app window is square, this means that the ball is perfectly round.

To modify the app window height, the *height* line in *setup.agc* needs to changed to

```
height=1024
```

When the height of the app window is changed, 10% of the height is much greater than 10% of the width and so the ball becomes stretched.

Activity 6.3

The line

```
SetSpriteSize(1,10,10)
```

should first be changed to

```
SetSpriteSize(1,-1,10)
```

The ball will be round but this time it is 10% of the height and so, much larger than previously.

On the next run the line should now read

```
SetSpriteSize(1,10,-1)
```

which will return the ball to the size it had been before we resized the app window (10% of the width).

Activity 6.4

Modified *FirstSprite*:

```
rem *** First Sprite ***

rem *** Clear screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Resize sprite ***
SetSpriteSize(1,10,-1)
Sync()
do
loop
```

The black pixels are invisible.

Activity 6.5

Modified *FirstSprite*:

```
rem *** First Sprite ***

rem *** Clear screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Resize sprite ***
SetSpriteSize(1,10,-1)
Sync()
rem *** Wait then reposition sprite ***
Sleep(2000)
SetSpritePosition(1,50,50)
Sync()
do
loop
```

Activity 6.6

Modified *FirstSprite*:

```
rem *** First Sprite ***

rem *** Clear screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load image ***
LoadImage(1,"ball.bmp",1)
rem *** Create sprite ***
CreateSprite(1,1)
rem *** Resize sprite ***
SetSpriteSize(1,10,-1)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
    Sleep(50)
next p
do
loop
```

Activity 6.7

No solution required.

Activity 6.8

Modified *FirstSprite*:

```
rem *** Sprite Depth ***

rem *** Change screen to white ***
SetClearColor(255,255,255)
Sync()

rem *** Load images ***
LoadImage(1,"ball.bmp",1)
LoadImage(2,"poppy.bmp",1)
rem *** Create ball sprite ***
CreateSprite(1,1)
rem *** Bring sprite forward ***
SetSpriteDepth(1,9)
SetSpriteSize(1,10,-1)
rem *** Create poppy sprite ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
    Sleep(50)
next p
do
loop
```

The ball passes in front of the poppy rather than behind it.

Activity 6.9

Modified *FirstSprite*:

```
rem *** Sprite Depth ***

rem *** Change screen to white ***
SetClearColor(255,255,255)
Sync()

rem *** Load images ***
LoadImage(1,"ball.bmp",1)
LoadImage(2,"poppy.bmp",1)
rem *** Create ball sprite ***
CreateSprite(1,1)
rem *** Bring sprite forward ***
SetSpriteDepth(1,9)
SetSpriteSize(1,10,-1)
rem *** Create poppy sprites ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
CloneSprite(3,2)
SetSpritePosition(2,20,20)
rem *** Move cloned sprite to layer 8 ***
SetSpriteDepth(3,8)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
    Sleep(50)
next p
do
loop
```

The ball passes under the new poppy and over the original poppy.

Activity 6.10

Modified *FirstSprite*:

```
rem *** Sprite Hide ***

rem *** Change screen to white ***
SetClearColor(255,255,255)
Sync()
rem *** Load images ***
LoadImage(1,"ball.bmp",1)
LoadImage(2,"poppy.bmp",1)
rem *** Create ball sprite ***
CreateSprite(1,1)
rem *** Bring sprite forward ***
SetSpriteDepth(1,9)
SetSpriteSize(1,10,-1)
rem *** Create poppy sprites ***
CreateSprite(2,2)
SetSpriteSize(2,20,-1)
SetSpritePosition(2,50,50)
CloneSprite(3,2)
SetSpritePosition(2,20,20)
rem *** Move cloned sprite to layer 8 ***
SetSpriteDepth(3,8)
Sync()
rem *** Move sprite across the screen ***
for p = 1 to 100
    SetSpritePosition(1,p,p)
    Sync()
    Sleep(50)
next p
rem *** Hide poppies ***
SetSpriteVisible(2,0)
SetSpriteVisible(3,0)
Sync()
do
loop
```

Activity 6.11

The sound file *J1to10.wav* should play if everything is set up properly.

The sound file voices the numbers 1 to 10 in Japanese.

Activity 6.12

The text should be in sync with the spoken words. Although the speaker pauses, the sound plays continuously even while the `Sleep()` statement is being executed.

Activity 6.13

Modified *Sounds*:

```
rem *** Play sound file ***
rem *** Load file ***
LoadSound(1,"J1to10.wav")
rem *** Start playing file ***
PlaySound(1,100,1)
do
    Sync()
loop
```

Without the `Sync()` statement the file will play only once.

Activity 6.14

Code for *Music*:

```
rem *** Play music ***

rem *** Load music Files ***
LoadMusic(1,"TrackA.mp3")
LoadMusic(2,"TrackB.mp3")
LoadMusic(3,"TrackC.mp3")
rem ** Play last track once ***
PlayMusic(3,0,3,3)
do
loop
```

Activity 6.15

The messages will appear briefly as the mouse button is pressed and released.

Activity 6.16

The *Pressed* message remains visible while the mouse button is down; the *Released* message remains visible while the mouse button is up.

Activity 6.17

Modified *PressedFlower*:

```
Sync()
do
    if GetPointerState()=1
        PrintC(GetPointerX())
        PrintC(" ")
        Print(GetPointerY())
    else
        Print("Released")
    endif
    Sync()
loop
```

Activity 6.18

Modified *PressedFlower*:

```
rem *** Load image ***
LoadImage(1,"poppy.bmp")
rem *** Create sprite ***
CreateSprite(1,1)
SetSpritePosition(1,50,50)
SetSpriteSize(1,15,-1)
Sync()
do
    rem *** IF pointer pressed THEN ***
    if GetPointerPressed()=1
        rem *** Get its coordinates ***
        x# = GetPointerX()
        y# = GetPointerY()
        rem *** Check if coord over a sprite ***
        hit = GetSpriteHit(x#,y#)
        rem ***IF they are THEN hide sprite ***
        if hit <> 0
            SetSpriteVisible(1,0)
        endif
    endif
    Sync()
loop
```

Activity 6.19

Modified *UsingText*:

```
rem *** Text Resources demo ***

rem *** Create empty string ***
CreateText(1,"")
rem *** Set resource attributes ***
SetTextPosition(1,15,30)
SetTextColor(1,250,250,0,255)
SetTextSize(1,10)
rem *** FOR 5 times DO ***
for c = 1 to 5
    rem *** Empty string ***
    text$ = ""
    for dots = 1 to 10
        rem *** Add underscore to string ***
        text$ = text$+"_"
        rem *** Place string in text resource ***
        SetTextString(1,text$)

        Sync()
        rem *** Wait 200 msecs ***
        Sleep(200)
    next dots
    rem *** Empty text resource ***
    SetTextString(1,"")
    Sync()
    rem *** Wait one second ***"
    Sleep(1000)
next c
rem *** Delete resource ***
DeleteText(1)
Sync()
do
loop
```