7

Spot the Difference Game

In this Chapter:

- ❑ Designing Screen Layouts
- ❑ Creating Sprite Images
- ❑ Adding Background Music
- ❑ Adding Sound Effects
- ❑ Changing Screen Orientation
- ❑ Game Testing

Game - Spot the Difference

Introduction

At last, we know enough AGK BASIC to create a first game. This game is a 21st century update on the spot-the-difference game so beloved of many magazines. The game shows two almost identical images and the challenge is to spot the differences between the two images.

Game Design

When creating a game, there are many aspects of that game that we have to think about before we start to write program code.

Since this is a computer game derived from an existing paper-based one, we don't have to worry about giving an in-depth description of the game, defining the rules or stating how the game is won.

On the other hand, we still need to design the screen layout for the game. In fact, there may be several layouts to design: a start-up splash screen, the main game screen, an end-game screen and a credits screen detailing all those involved in the game development. Not only the overall screen designs need to be considered, but also the design of any individual sprites that may appear during the game play.

Any background music and sound effects not only have to be created, but when these are to be played also needs to be specified.

User interaction methods and help options are other aspects that have to be considered.

Game Description

In our game, the player is presented with two almost identical images. The left-hand image is the original image; the right-hand image has six modifications. The aim of the game is for the player to click (press) on the areas of the right-hand image that differ from those in the left-hand image.

The time elapsed since the start of the game is continually displayed.

The total time (in seconds) taken to find all six differences is displayed at the end of the game.

Screen Layouts

This game will have four screen layouts: splash screen, game screen, finish screen and credits screen.

You may want to create a rough drawing of the various screen layouts before going on to create a more detailed design using a drawing or paint package.

Another important point at this stage is to consider the screen size and resolution of the device(s) on which you want the game to run. Although AGK will allow your game to run on almost any platform, you may still want to consider how the screen size will affect the playability of your game. For example, 10 buttons along the right-hand edge of an iPad looks fine, but try the same thing on an iPhone and only the

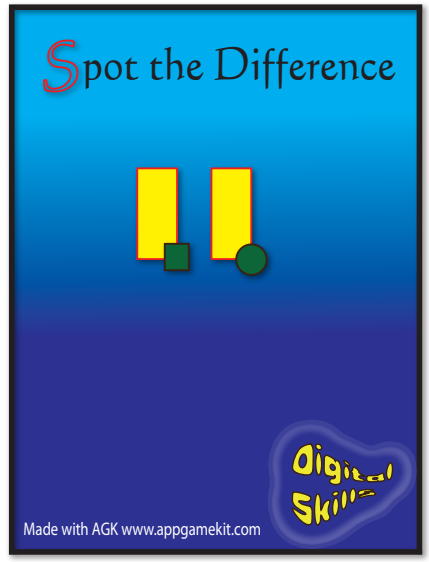
smallest of fingers will be able to use the buttons easily! And what about the near future? If you create images which are 1024 x 768 pixels in size with the iPad 2 in mind, what happens if a later iPad has a screen resolution of 2048 x 1536 pixels? Your images may not look as good on that!

For this game, the screen layouts have been designed using Adobe Illustrator which is a vector-drawing package. The great advantage of a vector-based image is that it can be converted to a regular bitmap image giving the best possible quality for a required resolution.

The splash screen (filename : AGK*Splash.png*) is shown in FIG-7.1.

FIG-7.1

The Splash Screen



This is a single PNG image. Note that it includes the name of the game, the company name (Digital Skills), text stating that it was built using AGK and the AGK website address. This last element is requested of you by **The Game Creators** if you are going to publish your app, but is not compulsory.

The second image (see FIG-7.2) is of the game screen containing the two photographs that form the game. This is the only image in landscape mode.

FIG-7.2

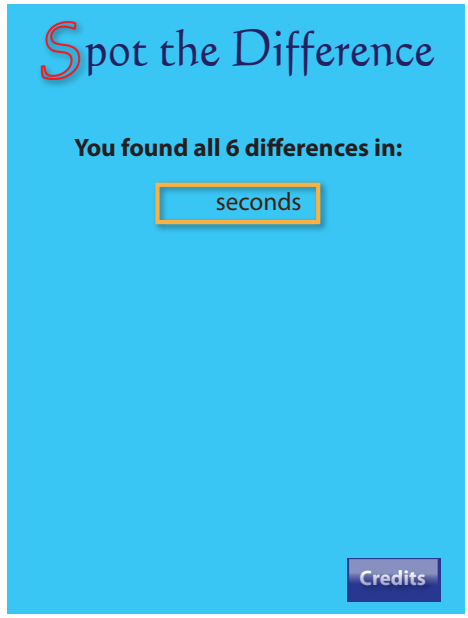
The Main Screen

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

The third image is the end screen which shows the total time taken in seconds (see FIG-7.3).

FIG-7.3

The End Screen

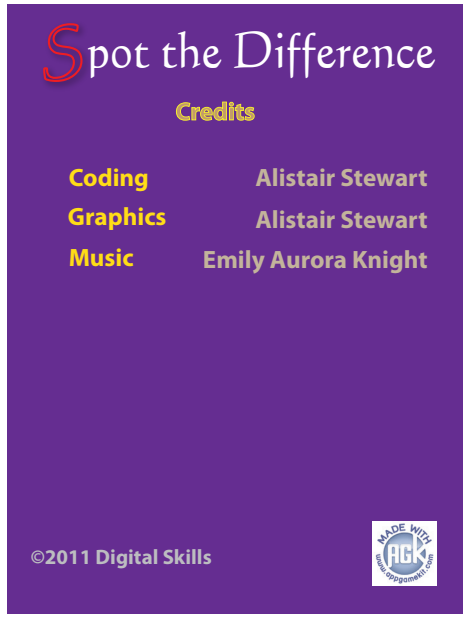


Again, you can see that a space has been left for the actual number of seconds taken to find all the differences. In addition, this screen also shows a separate button sprite in the bottom-right which allows the user to view the credits screen if required.

The final screen (see FIG-7.4) shows the names of those involved in creating the various aspects of the game: graphics, code, music. It also adds copyright details and the AGK logo.

FIG-7.4

The Credits Screen



A final visual component is the ring which appears around the differences in the photograph when the player presses in the correct area. Although there will be six of these, all make use of the same image (see FIG-7.5).

FIG-7.5

The Circle Spite

Other Resources

The only other resources used in the game are a sound effect, which plays when a modified area of the photo is pressed for the first time, and music which plays in the background while the game is running.

Overall Game Document

A useful document to produce is one showing not only the four screen layouts but also giving details of any sounds or actions that can occur during each stage of the game (see FIG-7.6).

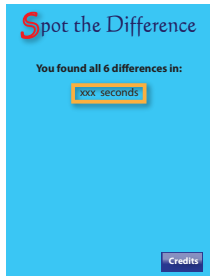
FIG-7.6

The Overall Game Document

Splash Screen

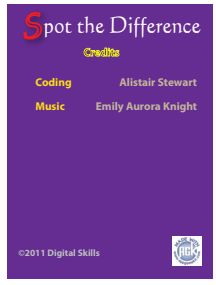
Main Screen

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



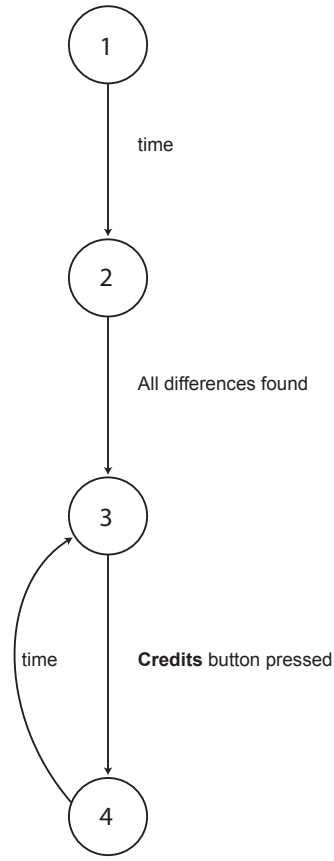
End Screen

- Music continues
- Displays total time taken to find all differences



Credits Screen

- Music continues



In the *Main* and *End* screen layouts X's are used to indicate where text is to be positioned, but the exact value of that text is unknown at the time of the design.

The *Main* screen is in landscape mode, while the other three screens are designed for portrait mode. As a general rule, it is best not to switch between modes within a game, but in this example it is interesting to see how the actual game play experience is affected by the transition.

On the right of FIG-7.6 is a **state-transition** diagram. The numbered circles represent the four different screen layouts. When each new screen appears during the game we consider the game to have entered a new **state**. The lines between the circles represent the moving from one state to another (i.e. from one screen to another). The text beside the lines explains what causes the game to move from one state to another. So we see that we move from the splash screen to the main game screen once an unspecified amount of time has passed; we move from the main screen to the end screen when all 6 differences have been found. Notice that we move to the credits screen only if the **Credits** button is pressed and that we return from the credits screen to the end screen after some time has elapsed.

For a more complex game, we might need to give greater detail for the design of each screen and the individual sprites which may appear on that screen.

Copyright Issues

Of course, if you intend to create a game simply for the amusement of yourself and your family, then making use of images you find on the internet, or adding your favourite music to the game isn't really a problem. However, should you wish to turn your game into a commercial product then you must make sure all aspects of the game are either copyright free, that you have permission from the copyright holder to use the material, or that the material is entirely of your own creation.

Even if you created the photographs used in a game, you can still breach copyright. For example, you can't use someone's image in a commercial product without their approval. You can't even use some buildings! If you were to use images taken in a Disney park for example, you would probably have their lawyers on your doorstep before you had made your first 10 sales!

Even if you record your own music, the melody itself may be copyrighted. Play and write your own music to be on the safe side.

You mustn't even borrow a one second sound effect without approval.

Don't worry! There are websites which offer copyright free material - but check that it can be used in a commercial product.

Finally, the images have no copyright problems, you have written and played the music, created all the sound effects, so you must be safe now, right? Afraid not! If you save your music in MP3 format, you'll find another set of lawyers wanting to have a few words. This time it won't happen until you've sold 5000 copies of your game but at that point you'll have to hand over large sums of money for the privilege of using the MP3 format. The way round this one is to use the OGG Vorbis format for your music files. AGK will automatically look for a file in this format even when your code specifies MP3.

And once you've made sure all your resources have no copyright issues, are you safe at last to write your game? Well, not entirely. You can still be on the receiving end of

a legal communication if someone thinks you've ripped off their game idea or even if your code makes use of some technique that has been copyrighted.

Have you given up all hope of creating a commercial game? Well, you can do a few things to protect yourself from the unexpected legal challenge. One option is to set up a limited company and publish your games through that (it's really not too complicated). Using this method, only your company can be sued if the worst should happen - not you. So you won't have to sell your home and flash new car to pay all the legal claims that have arrived on the doorstep.

And perhaps the easiest option of all is to let The Game Creators publish your game for you. Okay they are going to want 30%, but on the other hand they will test your game, suggest any changes, market it for you, even add revenue-gathering adverts and organise the cut-down free version and the paid-for full version. Chances are you'll sell more copies through them than you would do on your own and even after giving them their cut, you'll still make more money. And perhaps best of all, they are legally responsible - not you. Now, on with the game ...

Game Logic

The next stage is to do a high-level structured English description of the game.

The first level should be kept short:

```
1  Load resources
2  Set up game screen
3  Play game
4  End game
```

More detail can be added to each of these using stepwise refinement:

```
1 Load resources
    1.1 Load images
    1.2 Load sound
    1.3 Load music

2 Set up game screen
    2.1 Start music
    2.2 Display Main screen
    2.3 Add circles over differences
    2.4 Hide circles

3 Play game
    3.1 Start timer
    3.2 REPEAT
    3.3     IF user selected a difference THEN
    3.4         Show ring
    3.5         Play sound effect
    3.6     ENDIF
    3.7     Update time
    3.8 UNTIL all 6 differences selected
    3.9 Delete Main screen resources

4 End game
    4.1 Show End screen
    4.2 Display time taken
    4.3 Display Credits button
    4.4 DO
    4.5     IF Credits button pressed THEN
    4.6         Show Credits screen for 5 seconds
    4.7     ENDIF
    4.8 LOOP
```

Game Code

The game code follows the logic given above. The first section loads the resources but also includes comments on the overall program.

Structured English:

Load resources

Code:

```
rem ************************************
rem * program    : Spot the Difference  *
rem * version    : 1.0                  *
rem * language   : AGK BASIC v1.02      *
rem * date       : 18 Aug 2011          *
rem * author     : A. Stewart           *
rem * platform   : Ipad 1               *
rem ************************************

rem *** Load resources ***

rem *** Load images ***
main =    LoadImage("Main.jpg")
finish =  LoadImage("End.jpg")
credits = LoadImage("Credits.jpg")
ring =    LoadImage("Ring.png",0)
button =  LoadImage("Button.bmp",1)

rem *** Load sounds ***
ringsound = LoadSound("Click.wav")

rem *** Load music ***
backgroundmusic = LoadMusic("Background.wav")
```

Structured English:

Set up game screen

Code:

```
rem *** Play music ***
PlayMusic(BackgroundMusic)

rem *** Show main screen ***
CreateSprite(1,main)
SetSpriteSize(1,100,100)

rem *** Load rings at image differences ***
CreateSprite(2,ring)
SetSpriteSize(2,-1,10)
SetSpritePosition(2,91,86)
CloneSprite(3,2)
SetSpritePosition(3,51.5,22)
CloneSprite(4,2)
SetSpritePosition(4,49,68)
CloneSprite(5,2)
SetSpritePosition(5,73,66)
CloneSprite(6,2)
SetSpritePosition(6,88.5,66)
CloneSprite(7,2)
```

```
SetSpritePosition(7,55.75,62.5)

rem *** Hide rings ***
for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
next c
rem *** Update screen ***
Sync()
```

Structured English:

Play game

Code:

```
rem *** Start timer ***
start = GetSeconds()
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,0,255)
SetTextPosition(1,88,6)

rem *** Set count of differences found ***
found = 0
rem *** Get user clicks until all 6 differences found ***
repeat
    rem *** Check for clicked button ***
    pressed = GetPointerPressed()
    rem *** IF pressed, then check for sprite hit ***
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF clicked over hidden ring THEN
        if hit > 1 and hit <=7 and GetSpriteVisible(hit)=0
            rem *** Show ring ***
            SetSpriteVisible(hit,1)
            rem *** Play sound effect ***
            PlaySound(1)
            rem *** Add 1 to differences found ***
            found = found + 1
        endif
    endif
    rem *** Update time so far ***
    timetaken = GetSeconds() - start
    SetTextString(1,Str(timetaken))
    Sync()
until found = 6

rem *** Delete existing sprites ***
for c = 1 to 7
    DeleteSprite(c)
next c
rem *** Delete sound ***
DeleteSound(ringsound)
rem *** Delete text ***
DeleteText(1)
```

Note that we have had to add a *found* variable to keep count of how many differences have been found.

Structured English:

End game

Code:

```
rem *** Show End screen... ***
CreateSprite(1,finish)
SetSpriteSize(1,100,100)
rem *** ... with button... ***
CreateSprite(2,button)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,80,90)
rem *** ...and total time taken ***
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,0,255)
SetTextPosition(1,36,31)
Sync()

rem *** Allow for Credits button press ***
do
    pressed = GetPointerPressed()
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF Credit button pressed THEN ***
        if hit = 2
            rem *** Show credits screen for 5 seconds ***
            CreateSprite(3,credits)
            SetSpriteSize(3,100,100)
            SetSpriteDepth(3,8)
            Sync()
            Sleep(5000)
            rem *** Remove Credits screen ***
            DeleteSprite(3)
        endif
    endif
    Sync()
loop
```

The *Credits* screen is displayed "on top of" the *End* screen, so when it is deleted after 5 seconds, the *End* screen reappears.

Activity 7.1

Start a new project called *SpotTheDifference* and compile the default code so that the project's *media* folder is created.

From *AGKDownloads/Chapter7*, copy all the files in the folder to the project's *media* folder.

In *setup.agc* set width to 1024 and height to 768. This will create a landscape oriented app window.

Modify *main.agc* to match the code given over the last few pages. Test and save your code. What problems occurred?

No program is likely to be perfect on the first attempt. Perhaps there will be problems with the code: the logic may be wrong and this will be highlighted during testing.

The main problem with this first version of the game is caused by the fact that the main screen is in landscape mode but the *End* and *Credits* screens are in portrait mode. To get this to operate correctly, we need to change the screen orientation after the game is complete.

SetDisplayAspect()

We can change the screen's aspect ratio using the `SetDisplayAspect()` statement. In this statement we set the ratio of the width to the height. At the start of a program, the aspect ratio is determined by the values given for *width* and *height* in the *setup.agc* file. When the program is running, we can change to portrait orientation (but without changing the actual app window dimensions) using the line:

One of the numbers has to be real so that the calculation will produce a real (not integer) result.

```
SetDisplayAspect(768/1024.0)
```

The `SetDisplayAspect()` statement has the format shown in FIG-7.7.

FIG-7.7

SetDisplayAspect()

where:

value is a real number giving the ratio of the width to the height.

Activity 7.2

Modify your program so that, immediately after the resources of the main screen have been deleted, the display ratio is set using the lines:

```
rem *** Reset aspect ratio ***
SetDisplayAspect(768.0/1024.0)
```

Retest and save your program.

An important aspect to check is the finer details of game playability. For example, you may have noticed that when the last difference is found, the game jumps immediately to the *End* screen without giving the player a chance to see the placing of that final ring. We could solve this problem by getting the program to pause for one second before the *End* screen appears.

Activity 7.3

Add the lines

```
rem *** Wait before showing next screen ***
Sleep(1000)
```

immediately after the `DeleteText(1)` line.

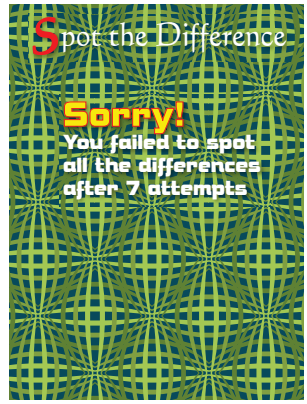
Test this modification and check that the player has time to see the final ring in position before the *End* screen appears.

A major problem with the game is that it has no way of stopping the player just pressing anywhere at random in the hope of hitting on a difference merely by chance. To stop this, we could introduce a maximum number of presses on the modified image. Perhaps 7 - this would allow the player one wrong attempt. However, introducing this change would mean that a new screen would have to be introduced

into the game, showing that the player had failed to complete the game. The *Failed* image is shown in FIG-7.8. This page will also show the **Credits** button.

FIG-7.8

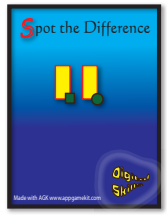
The Fail Screen



This modification to the program means that various parts of the game documentation also need to be changed. The first of these is the overall game document showing the various pages of the game and the state-transition diagram. The updated version of this document is shown in FIG-7.9.

FIG-7.9

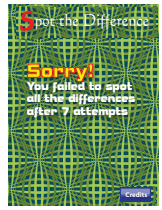
The Updated Game Document



Splash Screen

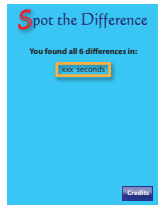
Main Screen

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



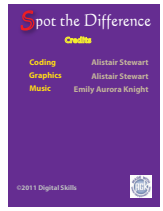
Failed Screen

- Music continues
- Displays failed message



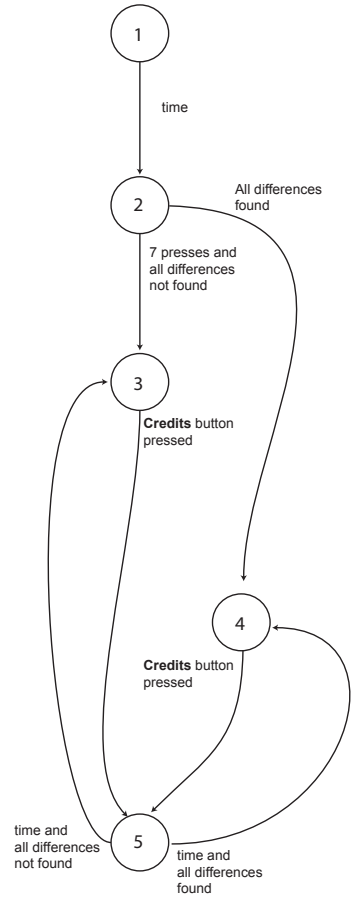
End Screen

- Music continues
- Displays total time taken to find all differences



Credits Screen

- Music continues



Note how much the state-transition diagram has changed. Not only have the state numbers assigned to the *End* and *Credits* screens changed, but the paths through the structure have become much more complex. From the *Main* screen (2) we may go to the *End* screen (4) if all 6 differences are found, but there is also an option to go to the *Fail* screen (3) when 7 presses have been made without all 6 differences being found. Both screens 3 and 4 have an option to show the *Credits* screen (5) for a set time period before screen 3 or 4 reappears. When the paths through the game start to become complex (as in this case), the state-transition diagram is a great way of maintaining an easy-to-follow overview of the whole game process.

The next part of the documentation to be changed is the structured English. Level 1 remains unchanged but the breakdown of some of its steps need to be modified. The updated logic is shown below with the changes highlighted.

```
3 Play game
    3.1 Start timer
    3.2 REPEAT
    3.3     IF user selected a difference THEN
    3.4         Show ring
    3.5         Play sound effect
    3.6     ENDIF
    3.7     Update time
    3.8 UNTIL all 6 differences selected or 7 presses made
    3.9 Delete Main screen resources


4 End game
    4.1 IF all 6 differences found THEN
    4.2     Show End screen
    4.3     Display time taken
    4.4 ELSE
    4.5     Show Fail screen
    4.6 ENDIF
    4.7 Display Credits button
    4.8 DO
    4.9     IF credits button pressed THEN
    4.10            Show Credits screen for 5 seconds
    4.11    ENDIF
    4.12 LOOP
```

Luckily, returning from the *Credits* screen to either the *End* or *Failed* screen isn't a problem since the *Credits* screen is shown on top of the previous screen. When the *Credits* screen is removed the appropriate screen will reappear.

Activity 7.4

Update your project to implement the changes described above. This requires the following steps:

Update the program's comments as appropriate.

- Copy the file *Fail.jpg* to the *media* folder.
- Add a line of code to load the image.
- The ID given to the image should be stored in the variable *fail*.
- Before the `repeat..until` loop, create a variable called *presscount* and set it to zero. Increment this variable every time `pressed = 1` is true.
- Add `or presscount = 7` to the condition in the `until` statement.
- Add the code for the new `if` statement described in the *End Game* structured English.

Check that the updated version of the program operates correctly by first winning a game and then losing one. Check that the *Credits* screen shows correctly in both cases. Resave your project.

Solutions

Activity 7.1

The *media* folder should contain the following files:

AGKSplash.png
Background.wav
Button.bmp
Click.wav
Credits.jpg
End.jpg
Main.jpg
Ring.png

The dimension setting lines in *setup.agc* should be changed to:

```
width=1024
height=768
```

The complete program code in *main.agc* is:

```
rem ************************************
rem * program    : Spot the Difference  *
rem * version    : 1.0                  *
rem * language   : AGK BASIC v1.02      *
rem * date       : 18 Aug 2011          *
rem * author     : A. Stewart           *
rem * platform   : Ipad 1               *
rem ************************************

rem *** Load resources ***

rem *** Load images ***
main =    LoadImage("Main.jpg")
finish =  LoadImage("End.jpg")
credits = LoadImage("Credits.jpg")
ring =    LoadImage("Ring.png",0)
button =  LoadImage("Button.bmp",1)

rem *** Load sounds ***
ringsound = LoadSound("Click.wav")

rem *** Load music ***
backgroundmusic = LoadMusic("Background.wav")

rem *** Play music ***
PlayMusic(BackgroundMusic)

rem *** Show main screen ***
CreateSprite(1,main)
SetSpriteSize(1,100,100)

rem *** Load rings at image differences ***
CreateSprite(2,ring)
SetSpriteSize(2,-1,10)
SetSpritePosition(2,91,86)
CloneSprite(3,2)
SetSpritePosition(3,51.5,22)
CloneSprite(4,2)
SetSpritePosition(4,49,68)
CloneSprite(5,2)
SetSpritePosition(5,73,66)
CloneSprite(6,2)
SetSpritePosition(6,88.5,66)
CloneSprite(7,2)
SetSpritePosition(7,55.75,62.5)

rem *** Hide rings ***
for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
next c
rem *** Update screen ***
Sync()

rem *** Start timer ***
start = GetSeconds()
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,0,255)
SetTextPosition(1,88,6)

rem *** Set count of differences found ***
found = 0

rem *** Get user clicks until all 6 differences
↳found ***
repeat
     rem *** Check for clicked button ***
     pressed = GetPointerPressed()
     rem *** IF pressed, check for sprite hit ***
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF clicked over hidden ring THEN
        if hit > 1 and hit <=7 and
          ↳GetSpriteVisible(hit)=0
            rem *** Show ring ***
            SetSpriteVisible(hit,1)
            rem *** Play sound effect ***
            PlaySound(1)
            rem *** Add 1 to differences found ***
            found = found + 1
        endif
    endif
    rem *** Update time so far ***
    timetaken = GetSeconds() - start
    SetTextString(1,Str(timetaken))
    Sync()
until found = 6

rem *** Delete existing sprites ***
for c = 1 to 7
    DeleteSprite(c)
next c
rem *** Delete sound ***
DeleteSound(ringsound)
rem *** Delete text ***
DeleteText(1)

rem *** Show End screen... ***
CreateSprite(1,finish)
SetSpriteSize(1,100,100)
rem *** ... with button... ***
CreateSprite(2,button)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,80,90)
rem *** ...and total time taken ***
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,0,255)
SetTextPosition(1,36,31)
Sync()

rem *** Allow for Credits button press ***
do
    pressed = GetPointerPressed()
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF Credit button pressed THEN ***
        if hit = 2
            rem *** Show credits for 5 secs ***
            CreateSprite(3,credits)
            SetSpriteSize(3,100,100)
            SetSpriteDepth(3,8)
            Sync()
            Sleep(5000)
            rem *** Remove Credits screen ***
            DeleteSprite(3)
        endif
    endif
    Sync()
loop
```

The main problem is that although the main screen appears correctly, the *End* and *Fail* screens are not positioned correctly.

Activity 7.2

The new lines (shown in bold) should be placed as follows:

```
rem *** Reset aspect ratio ***
SetDisplayAspect(768.0/1024.0)

rem *** Show End screen... ***
CreateSprite(1,finish)
SetSpriteSize(1,100,100)
```

This modification should ensure the *End* screen is correctly sized.

Activity 7.3

The new lines (shown in bold) should be placed as follows:

```
DeleteText(1)

rem *** Wait before showing next screen ***
Sleep(1000)

rem *** Show End screen ***
CreateSprite(1,finish)
```

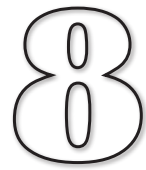
This gives a slight delay before the main screen disappears.

Activity 7.4

The file *Fail.jpg* should be added to the project's *media* file.

The final program code should be:

```
rem ************************************
rem * program    : Spot the Difference  *
rem * version    : 1.1                  *
rem * language   : AGK BASIC v1.02      *
rem * date       : 18 Aug 2011          *
rem * author     : A. Stewart           *
rem * platform   : Ipad 1               *
rem ************************************

rem *** Load resources ***

rem *** Load images ***
main =    LoadImage("Main.jpg")
finish =  LoadImage("End.jpg")
credits = LoadImage("Credits.jpg")
ring =    LoadImage("Ring.png",0)
button =  LoadImage("Button.bmp",1)
fail =    LoadImage("Fail.jpg")

rem *** Load sounds ***
ringsound = LoadSound("Click.wav")

rem *** Load music ***
backgroundmusic = LoadMusic("Background.wav")

rem *** Play music ***
PlayMusic(BackgroundMusic)

rem *** Show main screen ***
CreateSprite(1,main)
SetSpriteSize(1,100,100)

rem *** Load rings at image differences ***
CreateSprite(2,ring)
SetSpriteSize(2,-1,10)
SetSpritePosition(2,91,86)
CloneSprite(3,2)
SetSpritePosition(3,51.5,22)
CloneSprite(4,2)
SetSpritePosition(4,49,68)
CloneSprite(5,2)
SetSpritePosition(5,73,66)
CloneSprite(6,2)
SetSpritePosition(6,88.5,66)
CloneSprite(7,2)
SetSpritePosition(7,55.75,62.5)

rem *** Hide rings ***
for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
next c
rem *** Update screen ***
Sync()

rem *** Start timer ***
start = GetSeconds()
CreateText(1,Str(timetaken))
SetTextColor(1,0,0,0,255)
SetTextPosition(1,88,6)

rem *** Set count of differences found ***
found = 0
rem *** Number of clicks so far is zero ***
presscount = 0
rem *** Get user clicks until all 6 differences
↳found ***
repeat
     rem *** Check for clicked(pressed)
     pressed = GetPointerPressed()
     rem *** IF pressed,  ***
     if pressed = 1
        rem *** Add 1 to clicks ***
        inc presscount
        rem *** Check for sprite hit ***
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF clicked over hidden ring THEN
        if hit > 1 and hit <=7 and
          ↳GetSpriteVisible(hit)=0
            rem *** Show ring ***
            SetSpriteVisible(hit,1)
            rem *** Play sound effect ***
            PlaySound(1)
            rem *** Add 1 to differences found ***
            found = found + 1
        endif
    endif
    rem *** Update time so far ***
    timetaken = GetSeconds() - start
    SetTextString(1,Str(timetaken))
    Sync()
until found = 6 or presscount = 7

rem *** Delete existing sprites ***
for c = 1 to 7
    DeleteSprite(c)
next c
rem *** Delete sound ***
DeleteSound(ringsound)
rem *** Delete text ***
DeleteText(1)
rem *** Wait before showing next screen ***
Sleep(1000)
rem *** Reset aspect ratio ***
SetDisplayAspect(768.0/1024.0)
rem *** IF all differences found ***
if found = 6
   rem *** Show End screen... ***
   CreateSprite(1,finish)
   SetSpriteSize(1,100,100)
   rem *** ...and total time taken ***
   CreateText(1,Str(timetaken))
   SetTextColor(1,0,0,0,255)
   SetTextPosition(1,36,31)
else
   rem *** Show Fail screen... ***
   CreateSprite(1,fail)
   SetSpriteSize(1,100,100)
endif
Sync()
rem *** ... with button... ***
CreateSprite(2,button)
SetSpriteSize(2,15,-1)
SetSpritePosition(2,80,90)


rem *** Allow for Credits button press ***
do
    pressed = GetPointerPressed()
    if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF Credit button pressed THEN ***
        if hit = 2
            rem *** Show credits for 5 secs ***
            CreateSprite(3,credits)
            SetSpriteSize(3,100,100)
            SetSpriteDepth(3,8)
            Sync()
            Sleep(5000)
            rem *** Remove Credits screen ***
            DeleteSprite(3)
        endif
    endif
    Sync()
loop
```



User-Defined Functions

In this Chapter:

- ❑ Creating Functions
- ❑ **gosub** and **return** Statements
- ❑ Mini-Specs
- ❑ Modular Programming Concepts
- ❑ Parameter Passing
- ❑ Return Values
- ❑ Pre-Conditions

Functions

Introduction

Look at the computer in front of you. Notice how it is made up of several separate components: keyboard, screen, mouse, and inside the main casing are other discreet pieces such as the hard disk and CDROM.

Why are computers made this way, as a collection of separate pieces rather than having everything encased in a single frame?

Well, there are several reasons. Firstly, by using separate components, each can be designed to perform just one specific task such as: get information from the user (the keyboard); display information (the screen); store information (the disk) etc. This allows all of these items to be made and tested separately.

Also, if a component breaks down or needs to be replaced, you simply have to unplug that component and replace it with a new one.

Why is all of this relevant to creating games programs? Years of experience have shown that the advantages of this modular approach to construction doesn't just apply to physical items such as computers, it also applies to software.

Rather than create a program which consists of one continuous set of instructions, we can split the program into several **routines** (also known as **modules**, **functions** or **subroutines**). Each routine is designed to perform just one specific function. This approach is particularly important in long programs and when several programmers are involved in creating the software.

In fact, routines in AGK BASIC are usually referred to as **functions**, and that's the term we'll use from here on in.

Functions

Designing a Function

The first stage in creating a function is to decide what task the function has to perform. For example, we might want a function to do something as simple as display a line of asterisks or move a sprite about the screen.

A good function will perform only a single task and be relatively short - perhaps no more than 20 to 30 lines of code (often much less).

When a team of people is involved in creating the software, it is important that the exact purpose of each function is written out in detail so there can be no misconceptions between the people designing the routine and those programming it.

Functions must also be given a name. This name should reflect the purpose of the function and often starts with a verb, since functions perform tasks.

So let's have a first attempt at writing out a design for a function that is to draw a line of asterisks.

FUNCTION NAME	:	DrawLine
DESCRIPTION	:	Draws a horizontal line of 10 asterisks.

This function document is known as a **mini-spec** and, although it does not yet show all the features that will appear in a full mini-spec, it contains all the details we need to create our simple function. The only tricky part is to write a description that is unambiguous - something that is not always that easy! Notice the word *horizontal* has been included so that there is no possibility of the programmer deciding to create a function that produces a vertical line of asterisks.

Activity 8.1

List any other details that might be added to the description to make the requirements more exact.

Coding a Function

From the mini-spec we get the name and purpose of the function. From that we can create the following code:

```
function DrawLine()
   Print("**********")
endfunction
```

Notice that the module begins with the keyword `function` and ends with the keyword `endfunction`.

The first line also contains the name of the function, *DrawLine*, and an empty set of parentheses.

Between the first and last lines go the set of instructions that perform the task the function has been designed to do. In this case, only one line of code is needed.

Calling a Function

The code within a function will only be executed if that function is **called**. To call a function, a program need only specify the function's name and the empty parentheses:

```
DrawLine()
```

This is a request for the code within the named function to be executed.

The Final Code

The complete program will now contain two sections. One section will contain the code for the function and the other section the main logic of the program.

The function code must be placed after the end of the main logic.

This gives us the code shown in FIG-8.1.

FIG-8.1

Using a Function

```
rem *** main program ***

DrawLine()
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine()
    Print("**********")
endfunction
```

Notice that the `end` statement has been added to emphasise the end of the main program logic.

Activity 8.2

Start a new project called *UsingFunctions*. Modify *main.agc* to match the code given in FIG-8.1. Test and save your project.

How the Code is Executed

When a call is made to a function, control transfers to that function, its code is executed, and then control returns to the line immediately following the original call to the function. FIG-8.2 shows the stages involved during the execution of the program shown above.

FIG-8.2

The Function Calling Mechanism

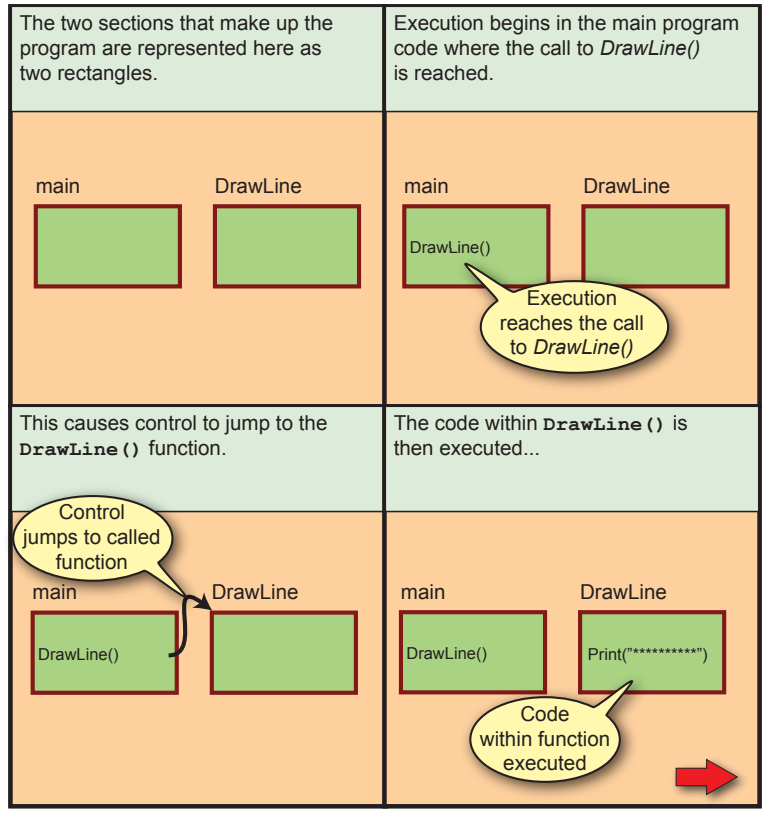
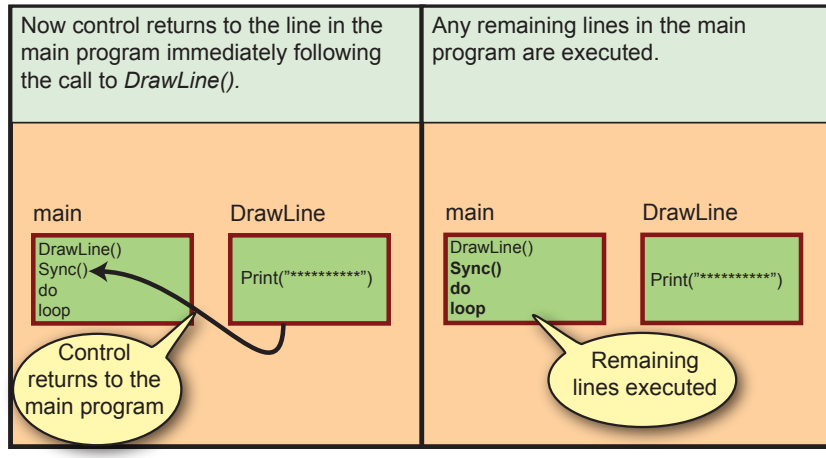


FIG-8.2
(continued)

The Function Calling Mechanism



A function can be called as often as required from any point in the main program logic.

So, if we wanted to draw two lines of asterisks, we would change the code for the main section to

```
DrawLine()
DrawLine()
Sync()
do
loop
end
```

which adds a second call to the function.

Activity 8.3

Modify *UsingFunctions* so that two calls are made to *DrawLine()* as in the code shown above.

Test your program to check that two lines of asterisks are drawn. Resave your project.

Local Variables

We are at liberty to use variables within a function. The variables used within a function are local to that function and hence are known as **local variables**. When a variable is local to a function, it is allocated space within the computer's memory only while that function is being executed. Once execution finishes, the allocated memory space is freed up and the variable no longer exists.

Because a variable is local to a function, that means that we may use a variable within a function that has the same name as a variable in the main program logic without causing an error. The program will treat the two variables as separate entities. The program in FIG-8.3 demonstrates this use of a local variable sharing a name with a variable in the main part of the program.

The variable *v* in the main program is assigned the value 3 while *v* within the function is assigned the value 6. The `Print` statement within the function will display the value held in the local variable (6) while the `Print` statement in the main section will

print 3 - the value in the other variable named *v*.

FIG-8.3

Using Local Variables

```
rem *** Variable v in the main program ***
v = 3
Test()
Print(v)
Sync()
do
loop
end

function Test()
   rem *** Variable v local to the function ***
   v = 6
   Print(v)
endfunction
```

Activity 8.4

Create a new project called *LocalVariable* and change *main.agc* to contain the code shown in FIG-8.3.

Run the program and verify that each variable contains a different value.

Alternative Coding

As long as a function performs the task described within the mini-spec, then exactly how that result is achieved is up to the programmer. Back on the first page of Chapter 1 we saw that there is usually more than one algorithm for achieving a required result (the 4 litre problem). So let us look at another way of creating a line of 10 asterisks:

```
function DrawLine()
   for c = 1 to 10
      PrintC("*")
   next c
   Print(" ") //New line
endfunction
```

If we take a moment to look at the code above, we can see that the `for` loop will print the 10 asterisks - one at a time - and the final `Print()` statement will move the cursor onto a new line.

Activity 8.5

Modify *UsingFunctions*, replacing the existing code for *DrawLine()* with the new code given above.

Check that exactly the same results are produced as before. Resave your project.

Parameters

Sometimes a device needs to be supplied with information before it can perform its function. For example, you need to press a button on your TV remote to specify which channel you want to view. This same principle also holds for software functions: most functions need to be supplied with one or more values in order to determine exactly what is required of it. These values are known as **parameters**.

If we wanted to allow the length of the line created by *DrawLine()* to be specified when the function is called, we need to pass that information to the function in the form of a parameter.

To pass a parameter to a function, we need to rewrite the description of that function adding parameter details such as the parameter name and its type. In the case of the *DrawLine()* function, the new mini-spec would be:

FUNCTION NAME	:	DrawLine
PARAMETERS		
In	:	ilength : integer
DESCRIPTION	:	Draws a horizontal line of *ilength* asterisks

Notice that the parameter is described as an **in parameter**. This description is used because the value is being "given" to the function.

From our updated description we create the new code:

The parameter given in the function's code is known as the **formal parameter.**

```
function DrawLine(ilength)
   for c = 1 to ilength
      PrintC("*")
   next c
   Print(" ")
endfunction
```

Notice that the parameter is placed within the parentheses of the first line of the function and that the parameter is then used as the end value in the `for` statement so that the loop now iterates *ilength* times.

And finally, the call made to the function from the main section of the program must supply a value for the parameter:

The parameter specified when the function is called is known as the **actual parameter.**

```
DrawLine(8)
```

This value will be copied into the function's formal parameter, *ilength*, just before the code of the function is executed.

Activity 8.6

In *UsingFunctions,* modify *DrawLine()* to match the code given above.

Change the calls to *DrawLine()* so that a line of 5 asterisks followed by a line of 12 asterisks is displayed. Test and save your project.

The actual parameter passed to a function can be given as a constant (as in the example above), a variable or an expression. Hence, if we start by storing a number in a variable

```
num = Random(1,20)
```

we can pass the value held in that variable as the parameter to our function with the line

```
DrawLine(num)
```

or we can include a calculation within the function call

```
DrawLine(num * 2)
```

and the result of that calculation will be passed as the parameter value.

Activity 8.7

Modify *UsingFunctions* so that the main program generates a random number in the range 1 to 10, storing the result in a variable, *num*.

Change the parameters given in the calls to *DrawLine()* so that the first call uses *num* as the parameter and the second call uses *num*3 -2* as the parameter.

Test and save your project.

A final option is to use the value returned by one function as the parameter for another as in the line:

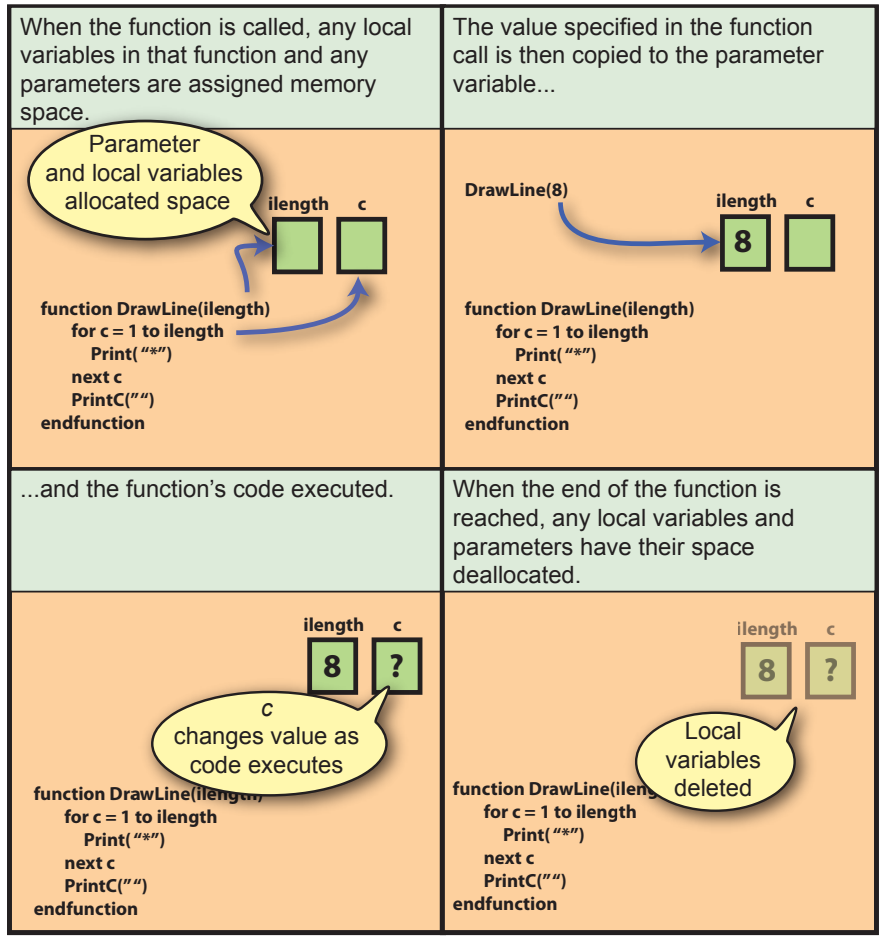
```
DrawLine(Random(1,10))
```

which will generate a random number in the range 1 to 10 and then use that value as the parameter to *DrawLine()*.

FIG-8.4 shows what's happening when a parameter is passed to a function.

FIG-8.4

How Parameter Passing Works



Multiple Parameters

A function can have as many parameters as required and these parameters can be of any type: integer, real or string.

To demonstrate this, we'll write yet another version of the *DrawLine()* function in which the character used to construct the line is also passed as a parameter. Of course, we start by updating the mini-spec:

FUNCTION NAME	:	DrawLine
PARAMETERS		
In	:	ilength : integer
	:	schar : string
DESCRIPTION	:	Draws a horizontal line of *ilength* characters. The character used in the construction of the line is *schar*.

From this we can create the modified function:

```
function DrawLine (ilength, schar$)
   for c = 1 to ilength
      PrintC(schar$)
   next c
   Print(" ")
endfunction
```

Notice that parameters are separated from each other by commas. The second parameter's name differs slightly from that in the mini-spec because of AGK BASIC's requirement that string variables must end with a dollar symbol ($).

The $ symbol is omitted from the parameter name since this is a BASIC requirement and not part of the design.

Now we need to supply two values when we call the function; one for the length of the line, the other for the character to be used in the construction of the line. A typical call would be:

```
DrawLine (12, "=")
```

which would assign the value 12 to the parameter *ilength* and the string "=" to *schar$* and thereby produce a line on the screen created from twelve = symbols.

Activity 8.8

Modify your *DrawLine()* function so that the character used can be passed as a parameter.

Change the main section of the code so that a single line of 10 # characters is drawn.

It's important that you put the values in the correct order when you call up a function. For example, the line

```
DrawLine ("=" ,12)
```

would be invalid since the function expects the first value given in the parentheses to be an integer.

Pre-conditions

When a function uses a parameter, we will often need to restrict the range of values which may sensibly be assigned to that parameter. For example, when we specify what length of line we want *DrawLine()* to produce, it doesn't make sense to pass a negative value to the function. Equally, it makes little sense to request a line hundreds of characters long, since there is a limit to how many characters can fit on a single line of the app window. We might therefore expect the value specified for *ilength* to be in a range such as 1 to 100.

When we place limitations on the conditions under which a function can operate successfully, these limitations are known as the **pre-conditions** of the function.

We can therefore state that the pre-condition for *DrawLine()* is that the parameter *ilength* lies between 1 and 100.

We would start by adding this restriction to the mini-spec:

FUNCTION NAME	:	DrawLine
PARAMETERS		
In	:	ilength : integer schar : string
PRE-CONDITION	:	1 <= *ilength* <= 100
DESCRIPTION	:	Draws a horizontal line of *ilength* characters. The character used in the construction of the line is *schar*.

Now we need some way of enforcing this limit. We do this by adding an `if` statement at the beginning of the *DrawLine()* function which causes that function to be aborted if the value of *ilength* is outside the acceptable range.

exitfunction

The `exitfunction` statement is designed to be placed within a function. When executed, this statement causes the remaining statements in the function to be ignored, ending execution of the routine and returning to the code which called the function in the first place.

We can make use of this statement to terminate execution of a function when its parameter(s) fall outside an acceptable range. In the case of *DrawLine()*, we would add the following lines right at the start of the routine to check the value of *ilength*:

```
rem *** if ilength outside 1 to 100, terminate function ***
if ilength < 1 or ilength > 100
   exitfunction
endif
```

Activity 8.9

Modify your *DrawLine()* function so that if *ilength* is outside the range 1 to 100, the function terminates without anything being drawn.

Check that the new code works by attempting to draw a line 101 characters long constructed from the + symbol.

About Pre-Conditions

Be careful when choosing the pre-condition for a function. A pre-condition should only prevent situations which the function itself cannot handle. Do not create a pre-condition simply because you feel that using a value outside a range seems unreasonable to you. If the function's code can handle the situation then allow that situation to occur.

For example, we have stated in the pre-condition of *DrawLine()* that *ilength* must be in the range 1 to 100. But could the code handle values outside that range? The answer to this question is - yes.

Values of zero or less will simply mean that the `for` loop will iterate zero times and so the only output will be that caused by the `Print(" ")` statement which will move subsequent output onto the next line on the screen. Values greater than 100, the characters produced may spread over several lines, but the function will still work as described.

So we were probably mistaken to impose a pre-condition on *DrawLine()*.

When a function has no restrictions we would describe the pre-condition within the mini-spec as:

```
PRE-CONDITION    :    None
```

Return Types

Not only can we supply values to a function (in the form of parameters), but some functions also return results.

For example, let's assume we wish to create a function called *SumIntegers()* which takes an integer parameter, *ival*, and returns the sum of all the integer values between 1 and *ival*.

When describing such a routine in a mini-spec, we add the returning value as an **out parameter**.

```
FUNCTION NAME   :   SumIntegers
PARAMETERS
   In           :   ival       : integer
   Out          :   iresult    : integer

PRE-CONDITION   :   None

DESCRIPTION     :   Sets iresult to the sum of the integers between 1
                    and ival.
```

To return a value from an AGK BASIC function we add the value to be returned immediately after the term `endfunction`. So *SumIntegers()* would be coded as:

```
function SumIntegers(ival)
   iresult = 0
   for c = 1 to ival
      iresult = iresult + c
   next c
endfunction iresult
```

Activity 8.10

Create a project called *TestFact* which contains a function named *Factorial* which implements the following mini-spec:

FUNCTION NAME	:	Factorial	
PARAMETERS			
In	:	ival	: integer
Out	:	iresult	: integer
PRE-CONDITION	:	None	
DESCRIPTION	:	Sets *result* to the product of the integers between 1 and *ival*. For example, if *ival* is 5 then *iresult* would be the value of 1 x 2 x 3 x 4 x 5.	

The main program should generate a random number between 1 and 10, display that number, then display the result of *Factorial()* using the generated value as the *In* parameter.

Test and save your project.

When calling a function that returns a value, that value can be assigned to a variable, displayed, or used in an expression. Examples of valid calls to *SumIntegers()* are given below:

```
sum = SumIntegers(10)
Print(SumIntegers(5))
answer = SumIntegers(9)/3
if SumIntegers(no) < 100
```

Functions can also return a string value. For example, the function *FillString(ch$, num)* returns a string containing *num* copies of *ch$*:

```
function FillString(ch$,num)
    sresult$ = ""
    for c = 1 to num
        sresult$ = sresult$ + ch$
    next c
endfunction sresult$
```

and might be called with a line such as

```
h$ = FillString("H",10)
```

which would place a string containing 10 H's in the variable *h$*.

Return Values and Pre-Conditions

Routines such as *SumIntegers()* and *Factorial()* cannot successfully return a result for all integer values, since the space assigned to a variable is of a fixed size so there would be insufficient space to hold the result produced by *SumIntegers(100)* or *Factorial(16)*. For this reason we need to impose a pre-condition in each case limiting the value of the In parameter.

Of course, this is easily done in the mini-spec. We could add the line

PRE-CONDITION : ival <= 50

in *SumIntegers*

and

PRE-CONDITION : 1 <= ival <= 15

in *Factorial*.

The problem arises when we attempt to implement these restrictions in the code of the functions. We might be tempted to start *SumIntegers()* with the lines

```
if ival > 50
   exitfunction
```

but because *SumIntegers()* is designed to return a value, it is not legal to exit that function without returning a value.

This means that the `exitfunction` statement, as shown above, is not valid since it attempts to exit the function without returning a value. Luckily, the statement's format allows for a value to be specified after the keyword `exitfunction` and this value is returned by the function.

But that just leaves us with another problem - what value should we return when the routine does not meet its pre-conditions? We can return any value we like, but usually this is handled by returning a special value which cannot occur when the function's pre-conditions are met. For example, here we could return the value -1, since it is an impossible result to achieve when *ival* is less than 50. We would do this with the line:

```
exitfunction -1
```

This allows us to add back the pre-condition to our routine, the final version of the code being:

```
function SumIntegers(ival)
   rem *** Exit with -1 if pre-condition ***
   rem *** not met                       ***
   if ival > 50
      exitfunction -1
   endif

   iresult = 0
   for c = 1 to ival
      iresult = iresult + c
   next c
endfunction iresult
```

Activity 8.11

Modify your *Factorial()* function from the last Activity so that it implements the pre-condition that *ival* must lie between 1 and 15. The function should return zero if the parameter is outside this range.

Test the update by calling the function with the parameter value set to 16. Resave your project.

When a function such as *SumIntegers()* (which returns a dummy result if its pre-condition has not been met) is called, the main program must check that the function has performed correctly. This is done by making the main program check the value returned by the function. A typical piece of code for doing this is:

The parameter *number* is assumed to be a variable that has been assigned a value earlier in the program.

```
result = SumIntegers(number)
if result = -1
   Print("Could not calculate result")
else
   Print(result)
endif
```

Activity 8.12

In *TestFact*, change the main program to generate a number between 10 and 20. Attempt to find the factorial of the number generated, but if the number is over 15, display the message "Factorial too high to calculate." along with the generated value.
Run your program so that at least one run produces the error message.

Returning a string from a function is no more difficult than returning a numeric value.

The program in FIG-8.5 contains a function which returns a random-length string of random letters.

FIG-8.5

Random Length String Function

```
rem *** Generate string ***
text$ = RandomString()
rem *** Display string ***
Print(text$)
Sync()
do
loop


rem *** Generate a random-length string of random letters ***
function RandomString()
    rem *** Generate length for string ***
    size = Random(1,50)
    rem *** start with empty string ***
    sresult$ =""
    rem *** FOR size times ***
    for c = 1 to size
        rem *** Add new character to end of string ***
        sresult$ = sresult$ + Chr(Random(65,90))
    next c
    rem *** return the string generated ***
endfunction sresult$
```

This code makes use of an AGK function called `Chr()` which returns the character whose ASCII code matches the value of the parameter. More about this function in the next chapter.

Activity 8.13

Create a mini-spec for the function *RandomString()* given in FIG-8.5.

Create a new project called *StringFunction*. Edit *main.agc* to match the code given in FIG-8.5.

Test and save the project.

Function Flexibility

The more flexible a function, the more useful it is. For example, the final version of *DrawLine()* is much more flexible than the first, since it allows the length and construction character of the line to be specified when the function is called, whereas the first version could create only a line of exactly ten asterisks. With this added flexibility we could use the function to draw a simple bar chart for example - something not possible with the original version of the routine.

So, wherever possible, you should always try to add the maximum flexibility to any routine you create as long as this does not lead to over-complex code or unacceptable execution times.

We will add some flexibility to our *RandomString()* function with a new mini-spec:

```
FUNCTION NAME   :   RandomString
PARAMETERS
    In          :   ilength    : integer
    Out         :   sresult    : string

PRE-CONDITION   :   ilength = -1 or 1 <= ilength <= 50

DESCRIPTION     :   IF ilength = -1 THEN
                        sresult is a string of random capital letters of a
                        random length between 1 and 50
                    ELSE
                        sresult is a string of random capital letters
                        exactly ilength characters long
                    ENDIF
```

➽ Although a mini-spec may have a description written in structured English, this does not mean that the program must employ that logic to implement the routine.

Notice that the mini-spec's description makes use of structured English this time. A description can be written in any way you please; the only requirement is that it must be complete and unambiguous. A mini-spec is the document used by the programmer as a statement of exactly what a function must do, so that document must contain all the details required.

Activity 8.14

Modify the code for *RandomString()* in your *StringFunction* project so that it matches the mini-spec requirements given above. If *ilength* is not within the specified range, an empty string should be returned.

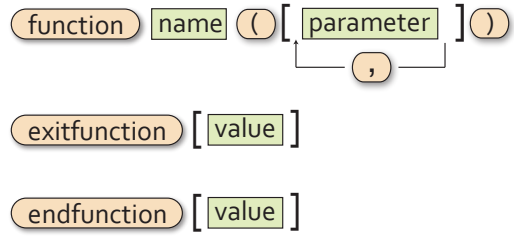
Test and save the project.

Statement Formats

We've introduced three new function-related statements in this section; the format of these are given in FIG-8.6.

FIG-8.6

function
exitfunction
endfunction



where:

name	is the name of the function. The name chosen must conform to the same rules used when creating variable names.
parameter	is the name of any value passed to the function. Names should be appropriate for the nature of the value being passed.
value	is the value returned by the function. This can be specified using a variable, constant or expression.

Summary

- A function is a named section of code.
- Functions should be relatively short and perform only a single task.
- Functions in AGK BASIC begin with the term `function` and end with the term `endfunction`.
- A function must be given a unique name.
- A function name should reflect the purpose of the function and must conform to the same rules as a variable name.
- A function can include zero or more *in* parameters.
- The parameter(s) listed in a function's code are known as the formal parameters.
- A function can return a single value.
- Any variables used within a function are local to that function.
- Local variables may have the same name as variables in the main program without causing an error.
- Before being coded, the details of a function should be documented in a mini-spec.
- Where a parameter's value must fall within a given range, this should be stated as a pre-condition in a function's mini-spec.
- Any pre-condition is tested by an `if` statement at the start of the function.
- When a pre-condition is not fulfilled, a function should exit without executing the main part of its code.
- Exiting a function before all of its code has been executed is achieved using the `exitfunction` statement.
- Where a pre-condition is not met and the function is designed to return a value, some error-indicating value should be returned.
- A function is called by giving the function name, parentheses and, where required, a list of parameter values.
- The parameters given when calling a function are known as the actual parameters.
- The value returned by a function can be assigned to a variable, displayed, used in an expression or used as the parameter to another function call.

BASIC Subroutines

Introduction

Using functions is the best way to create modular software in AGK BASIC, but the language does offer another way to achieve a similar effect, and that is to use subroutines. Although we've used the term subroutine earlier to mean any modular section of code, in AGK BASIC the word has a more specific meaning as we'll see below.

Creating a Subroutine

The original version of the BASIC language (invented in 1964) had no provision for true functions as described earlier in this chapter. Instead it made use of two statements, `gosub` and `return` which allowed a section of code to be executed and then a return made to the point of call. In this respect it was similar to a true function, but there was no way to pass parameter values or make use of local variables.

Although of limited usefulness, the `gosub` and `return` statements have been retained in AGK BASIC and so a description of how these statements are used is included here.

In order to compare true functions with the subroutine approach of `gosub`, we will recode the *DrawLine* routine using this older approach.

The start of a subroutine is marked with a label giving the name of the subroutine. This will be the name given in the mini-spec.

A label is just a valid name followed by a colon, for example:

```
DrawLine:
```

This is followed by the code

```
DrawLine:
    for c = 1 to 10
       Print("*")
    next c
    Print(" ")
```

and finally, the `return` statement:

```
DrawLine:
    for c = 1 to 10
       Print("*")
    next c
    Print(" ")
    return
```

To execute the code, we use the `gosub` statement giving the name of the label we used to start the code:

```
gosub DrawLine
```

A complete program implementing this example is shown in FIG-8.7.

FIG-8.7

Using Subroutines

```
rem *** Using GOSUB ***
gosub DrawLine
Sync()
do
loop
end

DrawLine:
    for c = 1 to 10
        PrintC("*")
    next c
    Print(" ")
    return
```

Activity 8.15

Start a new project called *TestGosub*.

Edit *main.agc* to match the code in FIG-8.7.

Test and save your project.

It is perhaps worth pointing out that no modern language offers this simplistic method of implementing modular programming because of the restrictions it imposes. Although you may see some examples of the `gosub` statement in action, these will be in very simple programs. So...

avoid using `gosub` - stick to proper functions!

A Library of Functions

Introduction

Most functions will be designed for a specific project and will only ever be used in that project, but a more general-purpose routine can be re-used in different programs.

We have already experienced this when we used the three Button functions back in earlier chapters to allow us to enter integer values.

Creating a Library

If we identify one or more routines that might be useful later, then we need to copy these routines into a new *agc* file which contains nothing but the code for these selected functions.

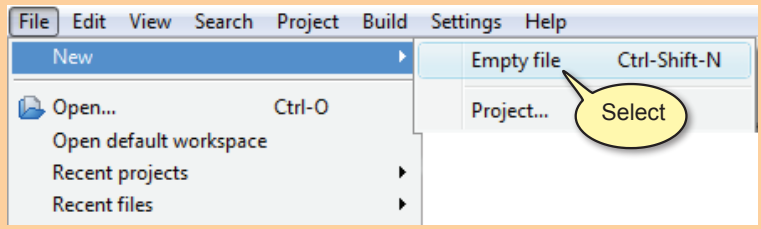
If you are building a library of reusable routines, the best approach is to create a separate *agc* file for each category. For example, we might keep all the math functions in one file and all the string-handling functions in another. Other functions which fall into an existing category can be added to the appropriate file later.

FIG-8.8 shows you how to place the *RandomString()* function in a separate file.

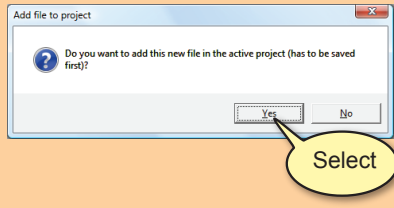
FIG-8.8

Creating a Library

Although the new Library file will be independant of any project, when it is first created it will be in the same folder as the function being transferred.
To add the new file, select **File|New|Empty** file from the main menu.



After requesting a new file we need to confirm that it is to be added to the current project.



Next we have to give the file a name. In this case, it has been called *StringLibrary* to remind us of its content and purpose.

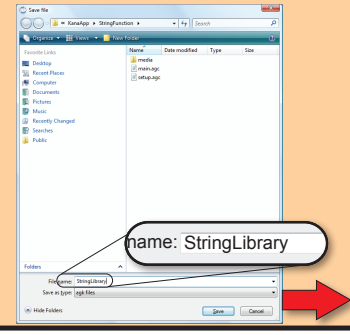
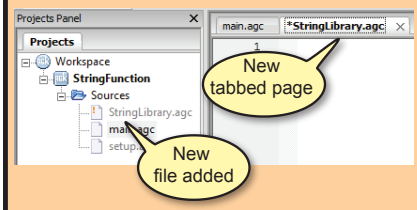
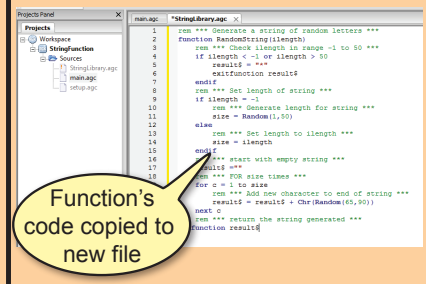


FIG-8.8
(continued)

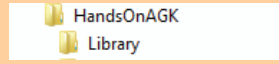
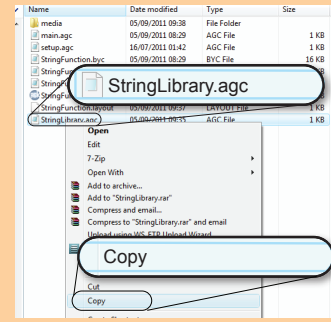
Creating a Library

The new file is now listed in the Projects Panel and has been assigned a tabbed page in the edit window.	Now we copy and paste the function's code to the new file before selecting the **File|Save Everything** from the main menu.
	

Since the point of creating this library file is to make use of the functions it contains in other projects, it makes sense to remove the file from the current project's folder and store it in a more generalised one (see FIG-8.9).

FIG-8.9

Moving the Library File

From Windows Explorer we need to create a folder for our library. Within the *HandsOnAGK* directory we create a new folder named *Library*.	Now we can copy the *StringLibrary* file from the *StringFunctions* project folder to our new *Library* folder.
	

Activity 8.16

Open your *StringFunction* project.

Create a new file called *StringLibrary*.

Copy the code for the function *RandomString()* from *main.agc* into the new file.

Save all the files within the project then close the project.

Open Windows Explorer and create a new subfolder called *Library* within the *HandsOnAGK* folder.

Copy the file *StringLibrary* from the *StringFunction* folder to the *Library* folder.

Creating Modular Software

Introduction

Now that we know the basic techniques required to design and implement functions in AGK BASIC, we're ready to rewrite the *SpotTheDifference* project using functions. We'll start by repeating the structured English description of the game:

1 Load resources
2 Set up game screen
3 Play game
4 End game

This is a good guide for identifying the routines needed within the program. The mini-spec for each identified routine is shown below.

```
FUNCTION NAME   :   LoadResources
PARAMETERS
    In          :   None

PRE-CONDITION   :   None

DESCRIPTION     :   The images
                        Button.bmp, Credits.jpg, End.jpg
                        Main.jpg, Ring.png, Fail.jpg
                    sound file
                        Click.wav
                    and music file
                        Backgroundmusic.wav
                    are loaded and assigned ID numbers.
```

```
FUNCTION NAME   :   SetUpGameScreen
PARAMETERS
    In          :   None

PRE-CONDITION   :   None

DESCRIPTION     :   Start music playing.
                    The image main.jpg is displayed. The rings are
                    positioned at each difference in the right-hand
                    picture and then hidden.
```

```
FUNCTION NAME   :   PlayGame
PARAMETERS
    In          :   None
    Out         :   timetaken : integer

PRE-CONDITION   :   None

DESCRIPTION     :   The user selects areas within the right-hand
                    picture. If the area is within a hidden ring, the ring
                    is displayed. The game ends when all six rings
                    are displayed or when 7 areas have been selected.
                    All resources used by this routine are deleted when
                    play is complete. The routine returns the number
                    of seconds taken to complete the game.
```

FUNCTION NAME	:	EndGame
PARAMETERS		
In	:	timetaken : integer
PRE-CONDITION	:	None
DESCRIPTION	:	Sets the aspect ratio to portrait mode. If the player has selected all six differences, then the *End* screen is displayed along with the time taken to find all the differences. If all differences were not found, the *Fail* screen is displayed. Both screens have a button which when pressed displays the *Credits* screen for 5 seconds before returning to the previous screen.

Now we're ready to start turning our design into a program.

There are various ways to tackle this. If we had several people working on the program, we could give each a separate routine to work on at the same time. It would then just be a matter of bringing together the separate routines to give us the final program. On the other hand, if only one person is working on the coding, the routines are coded one after another, usually starting with the main logic.

Top-Down Programming

When we code our routines one at a time, starting with the main logic, this is known as **top-down programming**. The name is used because we start with the main part of the program (the top) and then work our way through the routines called by that main part. We'll see how it's done below.

Step 1

We start by turning the outline logic given in the structured English into AGK BASIC code. An important point to note is that the program code must match that logic exactly. If we find we have to deviate from this logic, then we must go back and modify the details given in the structured English.

Actually, the code for the main section of the program becomes little more than a set of calls to the other routines:

```
LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
```

Notice that the main code really doesn't do any of the detailed work, it leaves that to the routines. The main section only has to call up each of the routines in the correct order and save any values returned by one function to pass it on to another function.

Activity 8.17

Start a new project called *SpotTheDifference2*.

Compile the default code and copy the required files into the *media* folder.
Modify *main.agc* so that it contains the four lines given above.

Edit *startup.agc* setting width to 1024 and height to 768.

Step 2

To get the main logic to run, we must write code for the routines that are called. And yet, if we do that, it would appear that the whole program will need to be completed before the program can be executed for the first time.

The way round this problem is to write almost empty routines with the required names as shown below:

```
function LoadResources
   Print("LoadResources")
endfunction

function SetUpGameScreen()
   Print("SetUpGameScreen")
endfunction

function PlayGame()
   Print("PlayGame")
endfunction 10

function EndGame(timetaken)
   Print("EndGame")
endfunction
```

Take a moment to look at this code. Each function displays its own name, takes any necessary parameters and returns a value where necessary. We need to make sure that the names, parameter names and return types match with those given in the mini-specs.

These empty routines are known as **test stubs** and are written so that we can test the main logic without having the final code for any of the routines which that logic has to call.

Activity 8.18

In *SpotTheDifference2*, add an `end` statement after the existing four lines of code. This will separate the main logic from the code for the functions.

Add the four test stubs given above to your program.

Add a `Sync()` statement and `do..loop` structure immediately before `end` so that the messages displayed by the function will appear on the screen.

Run and save your program.

By running the program, we can see that the functions are executed in the correct order.

Step 3

Now we can begin to remove the stubs in our project and replace them with the final version of each routine. As each new routine is added the program is tested to make sure that the new routine, and the program as a whole, are working correctly.

The code for the first routine, *LoadResources(),* is given below:

```
rem *** Load resources ***
function LoadResources()
    rem *** Load images ***
    main =      LoadImage("Main.jpg")
    finish =    LoadImage("End.jpg")
    credits =   LoadImage("Credits.jpg")
    ring =      LoadImage("Ring.png",0)
    button =    LoadImage("Button.bmp",1)
    fail =      LoadImage("Fail.jpg")
    rem *** Load sounds ***
    ringsound = LoadSound("Click.wav")
    rem *** Load music ***
    backgroundmusic = LoadMusic("Background.wav")
endfunction
```

Activity 8.19

Remove the *LoadResources()* test stub from your program and substitute the complete function as shown above.

Run the program again. Although no new output is produced, there will be an error message if any of the files cannot be found. Save your project.

The second routine, `SetUpGameScreen()`, is coded as:

```
rem *** Set up main section of the game ***
function SetUpGameScreen()
    rem *** Play music ***
    PlayMusic(backgroundmusic)
    rem *** Show main screen ***
    CreateSprite(1,main)
    SetSpriteSize(1,100,100)
    rem *** Load rings at image differences ***
    CreateSprite(2,ring)
    SetSpriteSize(2,-1,10)
    SetSpritePosition(2,91,86)
    CloneSprite(3,2)
    SetSpritePosition(3,51.5,22)
    CloneSprite(4,2)
    SetSpritePosition(4,49,68)
    CloneSprite(5,2)
    SetSpritePosition(5,73,66)
    CloneSprite(6,2)
    SetSpritePosition(6,88.5,66)
    CloneSprite(7,2)
    SetSpritePosition(7,55.75,62.5)
    rem *** Hide rings ***
    for c = 2 to 7
       SetSpriteDepth(c,9)
       SetSpriteVisible(c,0)
    next c
    rem *** Update screen ***
    Sync()
 endfunction
```

Activity 8.20

Add the complete version of *SetUpGameScreen()* to your project.

Run the program again. What problem occurs? Save your project.

Global Variables

The problem with *SetUpGameScreen()* is that it needs to make use of the IDs which were assigned to various resources by the *LoadResources()* function.

The *LoadResources()* function assigned the resource IDs to variables such as *backgroundmusic*, *main* and *ring*. But these variables are local to that routine, so when we mention variables of the same name in *SetUpGameScreen()* the program doesn't realise that we are trying to refer to the same variables as those in the earlier routine. Instead, we get a new set of variables that are local to *SetUpGameScreen()* and these new variables do not contain the values we need.

If only one value had been needed to be shared between the routines, we might have made use of a return value from the first routine and a parameter to the second (note that this is exactly how the *timetaken* is passed between *PlayGame()* and *EndGame()*).

However, since so many ID values need to be shared between *LoadResources()* and the other routines, we have no choice but to store these values in **global** variables.

Global variables are exactly the opposite from local variables. Whereas local variables exist only within the routine in which they are used, global variables exist throughout the program and can be referred to anywhere within the program.

To declare a global variable, we need to start with the keyword `global` and then give the variable name. In this program, we want the variables that contain the IDs of the various resources to be global - that way we can refer to them in any of the functions. So the code needed is:

```
rem *** Define global variables ***
rem *** IDs for images ***
global main, finish, credits, ring, button, fail
rem *** IDs for sound ***
global ringsound
rem *** IDs for music ***
global backgroundmusic
```

This code is placed right at the start of the main logic, before the function calls.

Now the term *main* in *LoadResources()* and *main* in *SetUpGameScreen()* refer to the same global variable.

Activity 8.21

Add the global declarations to your code and run the program again.

Does the program run correctly this time? Save your project.

The third function, *PlayGame()* is coded as:

```
rem *** Play game ***
function PlayGame()
    rem *** Start timer ***
    start = GetSeconds()
    CreateText(1,str(timetaken))
    SetTextColor(1,0,0,0,255)
    SetTextPosition(1,88,6)
    rem *** Set count of differences found ***
```

```
    found = 0
    rem *** Number of clicks so far is zero ***
    rem *** Get user clicks until all 6 differences found ***
    repeat
       rem *** Check for clicked(pressed)
       pressed = GetPointerPressed()
       rem *** IF pressed,  ***
       if pressed = 1
          rem *** Add 1 to clicks ***
          inc presscount
          rem *** Check for sprite hit ***
          x = GetPointerX()
          y = GetPointerY()
          hit = GetSpriteHit(x,y)
          rem *** IF clicked over hidden ring THEN
          if hit > 1 and hit <=7 and GetSpriteVisible(hit)=0
            rem *** Show ring ***
            SetSpriteVisible(hit,1)
            rem *** Play sound effect ***
            PlaySound(ringsound)
            rem *** Add 1 to differences found ***
            found = found + 1
          endif
       endif
       rem *** Update time so far ***
       timetaken = GetSeconds() - start
       SetTextString(1,Str(timetaken))
       Sync()
    until found = 6 or presscount = 7
    rem *** Delete existing sprites ***
    for c = 1 to 7
       DeleteSprite(c)
    next c
    rem *** Delete sound ***
    DeleteSound(ringsound)
    rem *** Delete text ***
    DeleteText(1)
endfunction timetaken
```

Activity 8.22

Add the third function to your project and check that it operates correctly.

Save your project.

The final function, *EndGame()* is coded as:

```
rem *** Finish game ***
function EndGame(timetaken)
    rem *** Wait before finishing ***
    Sleep(1000)
    rem *** Reset aspect ratio ***
    SetDisplayAspect(768.0/1024.0)
    rem *** IF all differences found ***
    if found = 6
       rem *** Show End screen... ***
       CreateSprite(1,finish)
       SetSpriteSize(1,100,100)
       rem *** ...and total time taken ***
       CreateText(1,str(timetaken))
       SetTextColor(1,0,0,0,255)
       SetTextPosition(1,36,31)
```

```
            Sync()
        else
            rem *** Show Fail screen... ***
            CreateSprite(1,fail)
            SetSpriteSize(1,100,100)
        endif
        rem *** ... with button... ***
        CreateSprite(2,button)
        SetSpriteSize(2,15,-1)
        SetSpritePosition(2,80,90)
        rem *** Allow for Credits button press ***
        do
            pressed = GetPointerPressed()
            if pressed = 1
                x = GetPointerX()
                y = GetPointerY()
                hit = GetSpriteHit(x,y)
                rem *** IF Credit button pressed THEN ***
                if hit =2
                    rem *** Show credits for 5 secs ***
                    CreateSprite(3,credits)
                    SetSpriteSize(3,100,100)
                    SetSpriteDepth(3,8)
                    Sync()
                    Sleep(5000)
                    rem *** Remove Credits screen ***
                    DeleteSprite(3)
                endif
            endif
            Sync()
        loop
    endfunction
```

Activity 8.23

Add the final function to your project and remove the `Sync()`, `do` and `loop` lines from the main section.

Test the completed project. Does it operate correctly?

The problem is that a variable whose value is determined in *PlayGame()* is required in *EndGame()*. What variable is this? To solve this problem, make the variable required in *EndGame()* a global variable.

Retest your project. Does it operate correctly? Save your project.

Activity 8.24

Now that the app is working correctly on your PC, it's time to try running it on another platform.

Make sure the app player or viewer is running on your device.

Press AGK's **Compile and Broadcast** button. The app should now start playing on your device. Does the app run correctly on the device?

The final problem we have to fix is to set the correct aspect ratio for the main game screen. Although we set the dimensions of the app window in *setup.agc*, when the app is running on your device, that setting has no relevance, so we need to add another `SetDisplayAspect()` statement to the *SetUpGameScreen()* function.

Activity 8.25

Modify the *SetUpGameScreen()* function so that it starts with the lines

```
rem *** Set screen aspect ***
SetDisplayAspect(1024.0/768)
```

Save the updated project.

Press the **Compile and Broadcast** button and check that the project now runs correctly.

Global Variables and Mini-Specs

As a general rule, we should try to avoid the use of global variables. Global variables make functions less independent of each other since those functions share access to the global variables. Global variables can also make finding bugs in a program more difficult since almost any of the routines could be assigning invalid values to those variables.

However, there are times when global variables will be necessary (as is the case with the resource IDs if we load all the resources in a single function).

When a program does contain global variables, then those global variables should be listed and described as part of the documentation along with the mini-specs.

```
GLOBAL VARIABLES in SpotTheDifference

Image IDs
    main, finish, credits, ring, button, fail     : INTEGER

Sound IDs
    ringsound                                     : INTEGER

Music IDs
    backgroundmusic                               : INTEGER

Number of differences found in image
    found                                         : INTEGER
```

Notice that the descriptions given give the purpose, name and type of any global variables.

Any routine that accesses global variables should include details of this. When a routine makes use of the current contents of a global variable, but does not change those contents, then the routine is said to **read** the variable. If a routine changes the contents of a global variable then this is known as a **write**.

Details of global variables read or written are added to a mini-spec after the parameter

details. So our updated mini-spec for *LoadResources* is:

```
FUNCTION NAME   :   LoadResources
PARAMETERS
    In          :   None

GLOBALS
    Read        :   None
    Written     :   button, credit, finish, main, ring, fail, ringsound,
                    backgroundmusic

PRE-CONDITION   :   None

DESCRIPTION     :   The images
                        Button.bmp, Credits.jpg, End.jpg
                        Main.jpg, Ring.png, Fail.jpg
                    sound file
                        Click.wav
                    and music file
                        Backgroundmusic.wav
                    are loaded and assigned ID numbers.
```

Activity 8.26

Update the other mini-specs for the *SpotTheDifference2* project to give details of any global variables referenced in each of the routines.

Bottom-Up Programming

Top-down programming is particularly suited to a single programmer working alone, but if you're working as part of a team of programmers, then you're likely to get landed with having to code a specific routine which, when completed, will be handed over to the team leader. He will then add your routine to the main program.

So let's assume we've just been landed with the job of writing the *EndScreen* function. How do we go about doing this task without having the other parts of the program?

Well, we need to start by getting hold of the mini-spec for the routine and turning it into a coded function.

Although we might be tempted to think our job is done when we have coded the function, we really need to check that our routine is operating correctly. It won't do your reputation as a programmer any good if you hand over code which contains obvious faults.

To test a function we start by making sure that what we've written conforms to the requirements of the mini-spec. Once we're happy with that, then the code itself must be tested. Since a function only executes when called by another piece of code, we need to write a main program which will call up the function we want to test. This main program, known as a **test driver**, needs to perform five main tasks:

- ➢ Set up any resources required by the function
- ➢ Supply a value for any parameters required by the function
- ➢ Execute the function
- ➢ Display the value of any parameters passed to the function

➢ Display any value returned by the function

Sometimes testing a function on its own is going to be difficult since it is so dependent on the existence of other functions. In the case of *EndScreen()* we need to make sure the appropriate images have been loaded and assign a value to the global variable *found*. The test driver for *EndScreen()* is shown in FIG-8.10.

FIG-8.10

EndGame() Test Driver

```
rem *** EndGame Test Driver ***

rem *** Set up required resources ***
rem *** Global variables required ***
global finish, credits, button, fail
global found
rem *** Images required ***
finish =     LoadImage("End.jpg")
credits =    LoadImage("Credits.jpg")
button =     LoadImage("Button.bmp",1)
fail =       LoadImage("Fail.jpg")

rem *** Set found ***
found = 6
rem *** Call function under test ***
EndGame(10)
end

rem *** Finish game ***
function EndGame(timetaken)
      rem *** Set screen aspect ***
      SetDisplayAspect(768/1024.0)
      rem *** IF all differences found ***
      if found = 6
         rem *** Show End screen... ***
         CreateSprite(1,finish)
         SetSpriteSize(1,100,100)
         rem *** ...and total time taken ***
         CreateText(1,str(timetaken))
         SetTextColor(1,0,0,0,255)
         SetTextPosition(1,36,31)
         Sync()
      else
         rem *** Show Fail screen... ***
         CreateSprite(1,fail)
         SetSpriteSize(1,100,100)
      endif
      rem *** ... with button... ***
      CreateSprite(2,button)
      SetSpriteSize(2,15,-1)
      SetSpritePosition(2,80,90)
      rem *** Allow for Credits button press ***
      do
         pressed = GetPointerPressed()
         if pressed = 1
         x = GetPointerX()
         y = GetPointerY()
         hit = GetSpriteHit(x,y)
         rem *** IF Credit button pressed THEN ***
         if hit =2
               rem *** Show credits for 5 secs ***
               CreateSprite(3,credits)
               SetSpriteSize(3,100,100)
               SetSpriteDepth(3,8)
```

FIG-8.10
(continued)

EndGame() Test Driver

```
                Sync()
                Sleep(5000)
                rem *** Remove Credits screen ***
                DeleteSprite(3)
            endif
        endif
        Sync()
    loop
endfunction
```

The coding shown here will test the routine working on the assumption that all 6 differences were found and that the total time taken was 10 seconds. To create a complete set of tests, we need to try various other times to ensure they are displayed correctly and also to set *found* to a value less than 6 which will check that the *Fail* screen is displayed correctly. Also, when the *End* and *Fail* screens are showing, we should press the **Credits** button to check that the *Credits* screen appears correctly.

Activity 8.27

Start a new project called *EndScreenTestDriver*.

Create the project's *media* folder and copy the relevant resources to the folder.

Change *main.agc* to match the code in FIG-8.10 and run the code.

Make changes to the code so that other times are used and that the *Fail* screen is displayed rather than the *End* screen.

While various programmers worked on creating the various routines of a project, the project leader would create the code for the main program, making use of a set of test stubs for the functions called. As each function became available from the rest of the team he would replace each test stub with the actual function code and test the program as each new routine was added.

This approach to program construction is known as **bottom-up programming**.

Structure Diagrams

As we begin to develop more complex programs containing several routines, it can be useful to retain an overview of the program's structure showing which routine is called by which, and the values that pass between them. This is done using a structure diagram.

A structure diagram contains one rectangle for each routine in a program, including a rectangle representing the main program code (this is given the name of the project). Each rectangle contains the name of the routine it represents. The collection of rectangles for the *SpotTheDifference2* project is shown in FIG-8.11.

FIG-8.11

Function Rectangles

In the design, the program is referred to as **SpotTheDifference** (the 2 being removed).

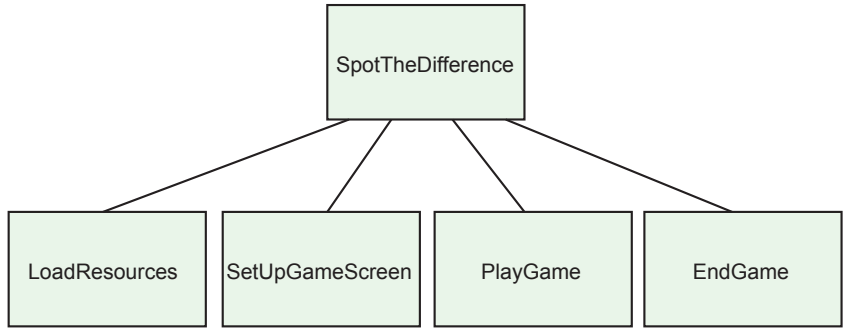
The rectangles are now set in a series of levels, with *SpotTheDifference* (the renamed

main logic) at the top. On the second level are routines called by *SpotTheDifference*.

The new layout is shown in FIG-8.12.

FIG-8.12

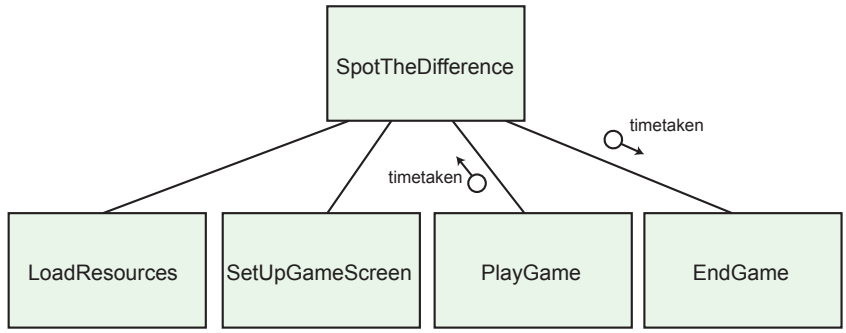
A Structure Diagram Showing Call Structure



Finally, we add any parameters passed between the routines. In this case *PlayGame* returns the time taken to find all the differences and *EndGame* takes that same value as an *In* parameter. Parameters are represented by labelled, directed circles (see FIG-8.13).

FIG-8.13

A Structure Diagram Showing Parameters



Note that global variables are not represented in structure diagrams.

The circle with arrowed line symbol in the diagram is used to show the direction in which data is passed, with *In* parameters pointing towards a routine and *Out* parameters pointing away from the routine.

Summary

- Good programming technique requires program code to be partitioned into routines.
- Each routine should perform a single task.
- A routine's name should reflect the purpose of that routine.
- Mini-specs should be produced when designing a routine.
- A mini-spec should include the name of the routine, its parameters, restrictions of the range of values a parameter may take and a detailed description of the routine's purpose.
- A routine should be made as flexible as possible so that it can be used in situations which differ slightly from the original requirement.
- The term `global` can be used to create a variable which can be accessed anywhere within a program.
- Top-down programming begins by coding the main routine.
- Top-down programming uses stubs in place of completed routines.

- Bottom-up programming starts by coding individual routines.
- Bottom-up programming uses test drivers to check that completed routines are operating correctly.
- A structure diagram shows every routine in a program, how they are called, and the values that pass between them.

Solutions

Activity 8.1

We could include the size, font and colour to be used for the asterisks.

Activity 8.2

No solution required.

Activity 8.3

Code for modified version of *UsingFunctions*:

```
rem *** main program ***
DrawLine()
DrawLine()
Sync()
do
loop
end


rem *** Draw a line function ***
function DrawLine()
    Print("**********")
endfunction
```

Activity 8.4

The program's output is

6
3

The first of these is the value of the variable *v* defined within the *Test()* function; the second is the value held in the main program's *v*.

Activity 8.5

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***
DrawLine()
DrawLine()
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine()
    for c = 1 to 10
        PrintC("*")
    next c
    Print(" ")
endfunction
```

Activity 8.6

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***
DrawLine(5)
DrawLine(12)
Sync()
do
loop
end


rem *** Draw a line function ***
function DrawLine(ilength)
    for c = 1 to ilength
        PrintC("*")
    next c
    Print(" ")
endfunction
```

Activity 8.7

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***

rem *** Generate random number ***
num = Random(1,10)
rem *** Call function ***
DrawLine(num)
DrawLine(num*3-2)
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine(ilength)
    for c = 1 to ilength
        PrintC("*")
    next c
    Print(" ")
endfunction
```

Activity 8.8

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***

rem *** Call function ***
DrawLine(10,"#")
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine(ilength, schar$)
    for c = 1 to ilength
        PrintC(schar$)
    next c
    Print(" ")
endfunction
```

Activity 8.9

Code for the updated version of *UsingFunctions*:

```
rem *** main program ***

rem *** Call function ***
DrawLine(101,"+")
Sync()
do
loop
end

rem *** Draw a line function ***
function DrawLine(ilength, schar$)
    if ilength < 1 or ilength > 100
        exitfunction
    endif
    for c = 1 to ilength
        PrintC(schar$)
    next c
    Print(" ")
endfunction
```

Activity 8.10

Code for *TestFact*:

```
rem *** main program ***
rem *** Generate random number ***
num = Random(1,10)
rem *** Find factorial of number generated ***
answer = Factorial(num)
rem *** Display results ***
PrintC("Factorial of ")
PrintC(num)
PrintC(" is ")
Print(answer)
Sync()
do
loop
end
```

```
rem *** Factorial Function ***
function Factorial(ival)
    iresult = 1
    for c = 2 to ival
        iresult = iresult * c
    next c
endfunction iresult
```

Activity 8.11

Code for the updated version of *TestFact*:

```
rem *** main program ***

answer = Factorial(16)
rem *** Display results ***
PrintC("Factorial of 16 is ")
Print(answer)
Sync()
do
loop
end


rem *** Factorial Function ***
function Factorial(ival)
    if ival < 1 or ival > 15
        exitfunction 0
    endif
    iresult = 1
    for c = 2 to ival
        iresult = iresult * c
    next c
endfunction iresult
```

Activity 8.12

Code for the updated version of *TestFact*:

```
rem *** main program ***
rem *** Generate random number ***
num = Random(10,20)
answer = Factorial(num)
if answer = 0
    rem *** Display error message ***
    PrintC(num)
    Print(" Factorial too high to calculate")
else
    rem *** Display results ***
    PrintC("Factorial of ")
    PrintC(num)
    PrintC(" is ")
    Print(answer)
endif
Sync()
do
loop
end

rem *** Factorial Function ***
function Factorial(ival)
    if ival < 1 or ival > 15
        exitfunction 0
    endif
    iresult = 1
    for c = 2 to ival
        iresult = iresult * c
    next c
endfunction iresult
```

Activity 8.13

FUNCTION NAME	:	RandomString
PARAMETERS		
In	:	None
Out	:	sresult : string
PRE-CONDITION	:	None
DESCRIPTION	:	Creates a string of random capital letters between 1 and 50 characters in length.

Activity 8.14

Code for the updated version of *StringFunction*:

```
rem *** Main program ***
rem *** Generate string ***
text1$ = RandomString(-1)
text2$ = RandomString(10)
text3$ = RandomString(-5)
rem *** Display strings ***
Print("Random length:" + text1$)
Print("Length 10    :" + text2$)
Print("Invalid      :" + text3$+"XXX")
Sync()
do
loop


rem *** Generate a random-length string of random
↳letters ***
function RandomString(ilength)
    rem *** IF invalid length, return empty string
    ↳***
    if ilength <>-1 and (ilength <1 or ilength >50)
        exitfunction ""
    endif
    rem *** Determine length of string ***
    if ilength = -1
        rem *** Generate length for string ***
        size = Random(1,50)
    else
        size = ilength
    endif
    rem *** start with empty string ***
    sresult$ = ""
    rem *** FOR size times ***
    for c = 1 to size
        rem *** Add new character to end of string
        ↳***
        sresult$ = sresult$ + Chr(Random(65,90))
    next c
    rem *** return the string generated ***
endfunction sresult$
```

Notice that the third `Print` statement in the main section adds XXX to the display. This is used to prove that the returned string is empty rather than filled with space characters. For an empty string, the final colon and XXX will be joined (:XXX); if the string contained spaces there would be a gap between the colon and the X's (: XXX).

Activity 8.15

No solution required.

Activity 8.16

No solution required.

Activity 8.17

Code for *SpotTheDifference2*:

```
LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
```

Changes to *setup.agc*:

```
width=1024
height=768
```

Activity 8.18

Code for the updated version of *SpotTheDifference2:*

```
LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end
```

```
function LoadResources
    Print("LoadResources")
endfunction

function SetUpGameScreen()
    Print("SetUpGameScreen")
endfunction

function PlayGame()
    Print("PlayGame")
endfunction 10

function EndGame(timetaken)
    Print("EndGame")
endfunction
```

You should see the names of the four functions appear when the program is run.

Activity 8.19

Code for the updated version of *SpotTheDifference2:*

```
LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end


rem *** Load resources ***
function LoadResources()
  rem *** Load images ***
  main =    LoadImage("Main.jpg")
  finish =  LoadImage("End.jpg")
  credits = LoadImage("Credits.jpg")
  ring =    LoadImage("Ring.png",0)
  button =  LoadImage("Button.bmp",1)
  fail =    LoadImage("Fail.jpg")
  rem *** Load sounds ***
  ringsound = LoadSound("Click.wav")
  rem *** Load music ***
  backgroundmusic = LoadMusic("Background.wav")
endfunction

function SetUpGameScreen()
    Print("SetUpGameScreen")
endfunction

function PlayGame()
    Print("PlayGame")
endfunction 10

function EndGame(timetaken)
    Print("EndGame")
endfunction
```

Activity 8.20

Code for the updated version of *SpotTheDifference2:*

```
LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end


rem *** Load resources ***
function LoadResources()
  rem *** Load images ***
  main =    LoadImage("Main.jpg")
  finish =  LoadImage("End.jpg")
  credits = LoadImage("Credits.jpg")
  ring =    LoadImage("Ring.png",0)
  button =  LoadImage("Button.bmp",1)
  fail =    LoadImage("Fail.jpg")
  rem *** Load sounds ***
  ringsound = LoadSound("Click.wav")
  rem *** Load music ***
  backgroundmusic = LoadMusic("Background.wav")
endfunction

rem *** Set up main section of the game ***
function SetUpGameScreen()
  rem *** Play music ***
  PlayMusic(backgroundmusic)
  rem *** Show main screen ***
  CreateSprite(1,main)
  SetSpriteSize(1,100,100)
  rem *** Load rings at image differences ***
  CreateSprite(2,ring)
  SetSpriteSize(2,-1,10)
  SetSpritePosition(2,91,86)
  CloneSprite(3,2)
  SetSpritePosition(3,51.5,22)
  CloneSprite(4,2)
  SetSpritePosition(4,49,68)
  CloneSprite(5,2)
  SetSpritePosition(5,73,66)
  CloneSprite(6,2)
  SetSpritePosition(6,88.5,66)
  CloneSprite(7,2)
  SetSpritePosition(7,55.75,62.5)
  rem *** Hide rings ***
  for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
  next c
  rem *** Update screen ***
  Sync()
endfunction

function PlayGame()
    Print("PlayGame")
endfunction 10

function EndGame(timetaken)
    Print("EndGame")
endfunction
```

The new function should begin by starting the background music. However, it fails when attempting to do this because it does not recognise the ID given for the music resource.

Activity 8.21

Code for the updated version of *SpotTheDifference2:*

```
rem *** Define global variables ***
rem *** IDs for images ***
global main, finish, credits,ring, button, fail
rem *** IDs for sound ***
global ringsound
rem *** IDs for music ***
global backgroundmusic

LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end

rem *** Load resources ***
function LoadResources()
  rem *** Load images ***
  main =    LoadImage("Main.jpg")
  finish =  LoadImage("End.jpg")
  credits = LoadImage("Credits.jpg")
  ring =    LoadImage("Ring.png",0)
  button =  LoadImage("Button.bmp",1)
  fail =    LoadImage("Fail.jpg")
  rem *** Load sounds ***
  ringsound = LoadSound("Click.wav")
  rem *** Load music ***
  backgroundmusic = LoadMusic("Background.wav")
endfunction

rem *** Set up main section of the game ***
function SetUpGameScreen()
  rem *** Play music ***
  PlayMusic(backgroundmusic)
  rem *** Show main screen ***
  CreateSprite(1,main)
  SetSpriteSize(1,100,100)
  rem *** Load rings at image differences ***
```

```
  CreateSprite(2,ring)
  SetSpriteSize(2,-1,10)
  SetSpritePosition(2,91,86)
  CloneSprite(3,2)
  SetSpritePosition(3,51.5,22)
  CloneSprite(4,2)
  SetSpritePosition(4,49,68)
  CloneSprite(5,2)
  SetSpritePosition(5,73,66)
  CloneSprite(6,2)
  SetSpritePosition(6,88.5,66)
  CloneSprite(7,2)
  SetSpritePosition(7,55.75,62.5)
  rem *** Hide rings ***
  for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
  next c
  rem *** Update screen ***
  Sync()
endfunction

function PlayGame()
    Print("PlayGame")
endfunction 10

function EndGame(timetaken)
    Print("EndGame")
endfunction
```

The program operates correctly now, getting as far as showing the main game screen.

Activity 8.22

Code for the updated version of *SpotTheDifference2:*

```
rem *** Define global variables ***
rem *** IDs for images ***
global main, finish, credits,ring, button, fail
rem *** IDs for sound ***
global ringsound
rem *** IDs for music ***
global backgroundmusic

LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
Sync()
do
loop
end

rem *** Load resources ***
function LoadResources()
  rem *** Load images ***
  main =    LoadImage("Main.jpg")
  finish =  LoadImage("End.jpg")
  credits = LoadImage("Credits.jpg")
  ring =    LoadImage("Ring.png",0)
  button =  LoadImage("Button.bmp",1)
  fail =    LoadImage("Fail.jpg")
  rem *** Load sounds ***
  ringsound = LoadSound("Click.wav")
  rem *** Load music ***
  backgroundmusic = LoadMusic("Background.wav")
endfunction

rem *** Set up main section of the game ***
function SetUpGameScreen()
  rem *** Play music ***
  PlayMusic(backgroundmusic)
  rem *** Show main screen ***
  CreateSprite(1,main)
  SetSpriteSize(1,100,100)
  rem *** Load rings at image differences ***
  CreateSprite(2,ring)
  SetSpriteSize(2,-1,10)
  SetSpritePosition(2,91,86)
  CloneSprite(3,2)
  SetSpritePosition(3,51.5,22)
  CloneSprite(4,2)
  SetSpritePosition(4,49,68)
  CloneSprite(5,2)
  SetSpritePosition(5,73,66)
  CloneSprite(6,2)
  SetSpritePosition(6,88.5,66)
  CloneSprite(7,2)
  SetSpritePosition(7,55.75,62.5)
  rem *** Hide rings ***
  for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
  next c
  rem *** Update screen ***
  Sync()
endfunction

rem *** Play game ***
function PlayGame()
  rem *** Start timer ***
  start = GetSeconds()
  CreateText(1,str(timetaken))
  SetTextColor(1,0,0,0,255)
  SetTextPosition(1,88,6)
  rem *** Set count of differences found ***
  found = 0
  rem *** Number of clicks so far is zero ***
  rem *** Get user clicks until all 6 differences
  ↳found ***
  repeat
    rem *** Check for clicked(pressed)
    pressed = GetPointerPressed()
    rem *** IF pressed,  ***
    if pressed = 1
      rem *** Add 1 to clicks ***
      inc presscount
      rem *** Check for sprite hit ***
      x = GetPointerX()
      y = GetPointerY()
      hit = GetSpriteHit(x,y)
      rem *** IF clicked over hidden ring THEN
      if hit > 1 and hit <=7 and
      ↳GetSpriteVisible(hit) = 0
        rem *** Show ring ***
        SetSpriteVisible(hit,1)
        rem *** Play sound effect ***
        PlaySound(ringsound)
        rem *** Add 1 to differences found ***
        found = found + 1
      endif
    endif
    rem *** Update time so far ***
    timetaken = GetSeconds() - start
    SetTextString(1,Str(timetaken))
    Sync()
  until found = 6 or presscount = 7
  rem *** Delete existing sprites ***
  for c = 1 to 7
    DeleteSprite(c)
  next c
  rem *** Delete sound ***
  DeleteSound(ringsound)
  rem *** Delete text ***
  DeleteText(1)
endfunction timetaken


function EndGame(timetaken)
    Print("EndGame")
endfunction
```

The program now allows the player to click on the six differences.

Activity 8.23

Code for the updated version of *SpotTheDifference2:*

```
rem *** Define global variables ***
rem *** IDs for images ***
global main, finish, credits,ring, button, fail
rem *** IDs for sound ***
global ringsound
rem *** IDs for music ***
global backgroundmusic

LoadResources()
SetUpGameScreen()
time = PlayGame()
EndGame(time)
end

rem *** Load resources ***
function LoadResources()
  rem *** Load images ***
  main =    LoadImage("Main.jpg")
```

```
  finish =   LoadImage("End.jpg")
  credits = LoadImage("Credits.jpg")
  ring =    LoadImage("Ring.png",0)
  button =  LoadImage("Button.bmp",1)
  fail =    LoadImage("Fail.jpg")
  rem *** Load sounds ***
  ringsound = LoadSound("Click.wav")
  rem *** Load music ***
  backgroundmusic = LoadMusic("Background.wav")
endfunction

rem *** Set up main section of the game ***
function SetUpGameScreen()
  rem *** Play music ***
  PlayMusic(backgroundmusic)
  rem *** Show main screen ***
  CreateSprite(1,main)
  SetSpriteSize(1,100,100)
  rem *** Load rings at image differences ***
  CreateSprite(2,ring)
  SetSpriteSize(2,-1,10)
  SetSpritePosition(2,91,86)
  CloneSprite(3,2)
  SetSpritePosition(3,51.5,22)
  CloneSprite(4,2)
  SetSpritePosition(4,49,68)
  CloneSprite(5,2)
  SetSpritePosition(5,73,66)
  CloneSprite(6,2)
  SetSpritePosition(6,88.5,66)
  CloneSprite(7,2)
  SetSpritePosition(7,55.75,62.5)
  rem *** Hide rings ***
  for c = 2 to 7
    SetSpriteDepth(c,9)
    SetSpriteVisible(c,0)
  next c
  rem *** Update screen ***
  Sync()
endfunction

rem *** Play game ***
function PlayGame()
  rem *** Reset aspect ratio ***
  SetDisplayAspect(768/1024.0)
  rem *** Start timer ***
  start = GetSeconds()
  CreateText(1,str(timetaken))
  SetTextColor(1,0,0,0,255)
  SetTextPosition(1,88,6)
  rem *** Set count of differences found ***
  found = 0
  rem *** Number of clicks so far is zero ***
  rem *** Get user clicks until all 6 differences
  found ***
  repeat
    rem *** Check for clicked(pressed)
    pressed = GetPointerPressed()
    rem *** IF pressed,  ***
    if pressed = 1
      rem *** Add 1 to clicks ***
      inc presscount
      rem *** Check for sprite hit ***
      x = GetPointerX()
      y = GetPointerY()
      hit = GetSpriteHit(x,y)
      rem *** IF clicked over hidden ring THEN
      if hit > 1 and hit <=7 and
      GetSpriteVisible(hit) = 0
        rem *** Show ring ***
        SetSpriteVisible(hit,1)
        rem *** Play sound effect ***
        PlaySound(ringsound)
        rem *** Add 1 to differences found ***
        found = found + 1
      endif
    endif
    rem *** Update time so far ***
    timetaken = GetSeconds() - start
    SetTextString(1,Str(timetaken))
    Sync()
  until found = 6 or presscount = 7
  rem *** Delete existing sprites ***
  for c = 1 to 7
    DeleteSprite(c)
  next c
  rem *** Delete sound ***
  DeleteSound(ringsound)
  rem *** Delete text ***
  DeleteText(1)
endfunction timetaken

rem *** Finish game ***
function EndGame(timetaken)
    rem *** Wait before finishing ***
    Sleep(1000)
    rem *** Reset aspect ratio ***
    SetDisplayAspect(768.0/1024.0)
    rem *** IF all differences found ***
    if found = 6
      rem *** Show End screen... ***
      CreateSprite(1,finish)
      SetSpriteSize(1,100,100)
      rem *** ...and total time taken ***
      CreateText(1,str(timetaken))
      SetTextColor(1,0,0,0,255)
      SetTextPosition(1,36,31)
      Sync()
    else
      rem *** Show Fail screen... ***
      CreateSprite(1,fail)
      SetSpriteSize(1,100,100)
    endif
    rem *** ... with button... ***
    CreateSprite(2,button)
    SetSpriteSize(2,15,-1)
    SetSpritePosition(2,80,90)
    rem *** Allow for Credits button press ***
    do
      pressed = GetPointerPressed()
      if pressed = 1
        x = GetPointerX()
        y = GetPointerY()
        hit = GetSpriteHit(x,y)
        rem *** IF Credit button pressed THEN ***
        if hit =2
            rem *** Show credits for 5 secs ***
            CreateSprite(3,credits)
            SetSpriteSize(3,100,100)
            SetSpriteDepth(3,8)
            Sync()
            Sleep(5000)
            rem *** Remove Credits screen ***
            DeleteSprite(3)
        endif
      endif
      Sync()
    loop
  endfunction
```

Even when we find all 6 differences the game shows the *Fail* screen.

The variable required is *found* which contains the count of how many differences were found by the player.

We need to add the code

```
rem *** Differences found ***
global found
```

The program works correctly after these lines have been added.

Activity 8.24

Although the app runs, it is not in landscape mode for the main screen.

Activity 8.25

The modified version of *SetUpGameScreen()*:

```
rem *** Set up main section of the game ***
function SetUpGameScreen()
  rem *** Set screen aspect ***
  SetDisplayAspect(1024.0/768)
  rem *** Play music ***
  PlayMusic(backgroundmusic)
  rem *** Show main screen ***
  CreateSprite(1,main)
  SetSpriteSize(1,100,100)
  rem *** Load rings at image differences ***
  CreateSprite(2,ring)
  SetSpriteSize(2,-1,10)
```

```
    SetSpritePosition(2,91,86)
    CloneSprite(3,2)
    SetSpritePosition(3,51.5,22)
    CloneSprite(4,2)
    SetSpritePosition(4,49,68)
    CloneSprite(5,2)
    SetSpritePosition(5,73,66)
    CloneSprite(6,2)
    SetSpritePosition(6,88.5,66)
    CloneSprite(7,2)
    SetSpritePosition(7,55.75,62.5)
    rem *** Hide rings ***
    for c = 2 to 7
      SetSpriteDepth(c,9)
      SetSpriteVisible(c,0)
    next c
    rem *** Update screen ***
    Sync()
endfunction
```

The game should now play correctly on your device.

Activity 8.26

FUNCTION NAME	:	SetUpGameScreen
PARAMETERS		
In	:	None
Out	:	None
GLOBALS		
Read	:	backgroundmusic, main, ring
Written	:	None
PRE-CONDITION	:	None
DESCRIPTION	:	Sets the aspect ratio of the screen to landscape, starts the background music playing, displays the main game screen, positions the rings over the image differences and hides the rings.

FUNCTION NAME	:	PlayGame
PARAMETERS		
In	:	None
Out	:	timetaken : integer
GLOBALS		
Read	:	ringsound
Written	:	found
PRE-CONDITION	:	None
DESCRIPTION	:	Allows the player to click on the screen up to 7 times. Keeps a count of how many have been found. Stops when all 6 found or when 7 clicks made.

FUNCTION NAME	:	EndGame
PARAMETERS		
In	:	timetaken : integer
Out	:	None
GLOBALS		
Read	:	found, finish, fail
Written	:	None
PRE-CONDITION	:	None
DESCRIPTION	:	The display returns to portrait layout. If all 6 differences found, the routine displays the Finish screen and the time taken in seconds. If all 6 are not found, the fail screen is displayed. If the Credits button is pressed, the Credits screen shows for 5 seconds.

Activity 8.27

To check that the *Fail* screen appears correctly, modify the line

```
found = 6
```

in the main section of the program to read

```
found = 5
```