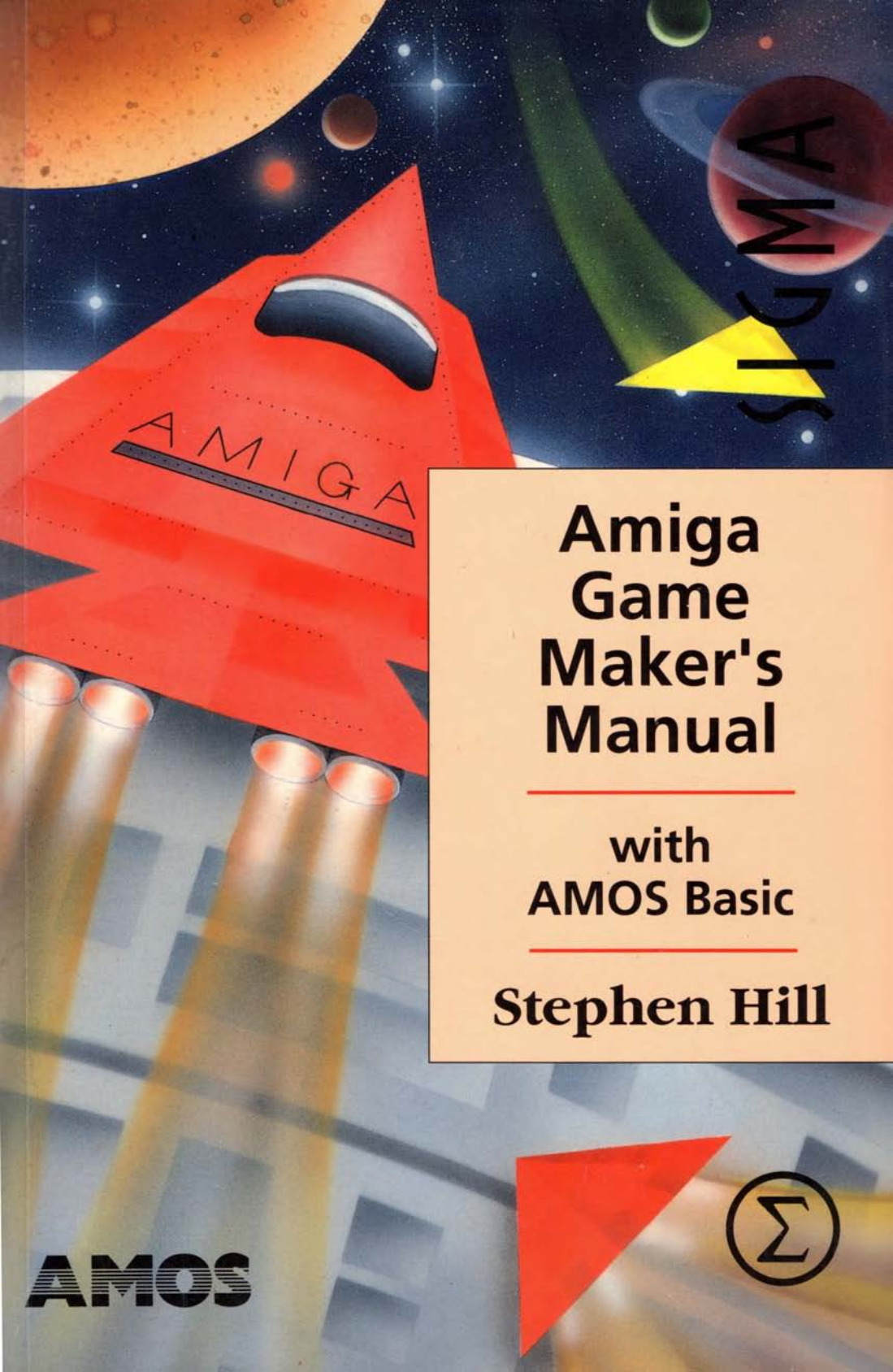
AMIGA
SIGMA
Amiga
Game
Maker's
Manual
with
AMOS Basic
Stephen Hill
AMOS

Amiga Game Maker's Manual

Stephen Hill

SIGMA PRESS
Wilmslow, England

Typeset and Designed by Sigma Hi-Tech Services Ltd, Wilmslow, UK

Cover Design by Design House, Marple Bridge.

First published in 1992
Sigma Press, 1 South Oak Lane, Wilmslow, Cheshire SK9 6AR, UK

First printed 1992
Reprinted 1993

ISBN: 1-85058-230-0

British Library Cataloguing in Publication Data
A CIP catalogue record for this book is available from the British Library

Printed by: Interprint Ltd, Malta

CONTENTS

1

First Steps

It's a well known fact that the journey of a thousand miles begins with just a single step. This is especially true if you're trying to write a computer game, as it's all too easy to fall flat on your face before you have really started. Wouldn't it be nice if there was someone out there to help you on your way? Well, that's the purpose of this book. It can't promise to solve all your problems for you, but I will certainly try to signpost some of the common pitfalls you may encounter during your journey.

1.1 Welcome to the Game Maker's Manual

With the advent of AMOS Basic, games creation is finally within the reach of even the most inexperienced Basic programmer. Forget about machine code! Who needs it? AMOS provides everything we need to produce commercial quality games on a standard A500 Amiga. The only essential ingredients are a general understanding of the Basic programming language, a healthy dollop of imagination, and a little hard work. That's it!

During the course of this book, I'll be presenting you with a detailed, step-by-step guide to creating anything from an ultra-fast arcade game to an adventure. It's going to be a long journey, but I'll try to ensure that there'll be plenty of interesting sights along the way. Have fun!

1.2 A Gentle Introduction to AMOS Graphics

When you're starting out, AMOS Basic can actually be pretty intimidating. There are literally hundreds of different commands to contend with, and it's easy to get bogged down with useless details. I'll therefore kick-off a complete introduction to the AMOS Basic graphics system. I'll intentionally concentrate on the specific instructions you'll need to use graphics successfully in your own programs. Once you've mastered the basic ideas, the rest of the commands should quickly fall into place. If you still get stuck, you'll now be able to refer back to the AMOS Basic manual with some confidence. Hopefully, this will prove invaluable if you've been struggling with the original manual! On the other hand, if you're already an old-hand at AMOS programming, feel free to jump directly the next chapter. Ok?

1.2.1 Coordinate systems

Let's start at the very beginning. If we want to draw something, we need some simple way of specifying its exact position on the screen. Take the AMOS PLOT command, for instance, which simply changes the colour of a single screen point (or pixel). We can't just type:

```
PLOT colour
```

as there are thousands of different points to choose from. So how can AMOS possibly know which one we are referring to? In practice, each point on the screen is located using a pair of numbers known as coordinates. These are measured from the top left hand corner of the current screen. This is the origin point.

Traditionally, the two coordinates are given the names X and Y. Don't ask me Y! The X coordinate holds the number of points from the right of our starting position. Similarly, the Y coordinate measures the vertical distance from the top of the screen to our chosen point. Whenever we enter our two coordinates into our instructions, we always separate them by a comma: X,Y. It's rather like the grid references of a map, with the first number giving the X and the second the Y coordinate. Figure 1.1 shows a few examples of these coordinates for you to look at.

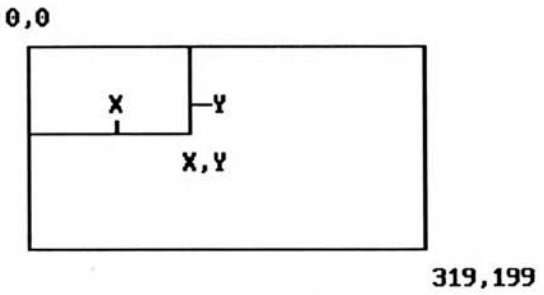

Figure 1.1: Screen Coordinates

- 0,0 are the coordinates of a point at the TOP LEFT corner of the current screen.
- 319,199 represents the location of a point at the bottom right of the standard display.
- 160,100 holds the approximate coordinates of a point at the centre of the screen.

Let's have a bash at using these coordinates from direct mode. Load up AMOS Basic, and jump to direct mode by pressing the ESC key. The familiar editor screen will slide away to be replaced by two large windows. The blue *direct mode* window is now waiting for us to enter our commands, and the orange *screen* window will be used for our graphics.

Let's plot a white dot at coordinates 160,100. The required instruction is:

```
AMOS>plot 160,100
```

Ignore the AMOS> bit, it's displayed automatically by AMOS Basic. We only need to type the letters after the > character. From hereon I will not include it.

The AMOS PLOT command comes in two flavours. The easiest version is just:

```
PLOT x,y
```

which plots a single point at coordinates x,y. Try replacing these coordinates with your own values. Before long, you'll able to position the dot anywhere on the screen.

1.2.2 Colours

The next stage is to add a little colour to the scene. The standard AMOS screen allows us to display up to 16 colours at any one time. Each colour is entered in AMOS Basic using a index ranging from 0 to 15. I'll demonstrate how this works using an extended version of the previous PLOT command. It has the form:

```
PLOT x,y,colour
```

where *colour* is an index number from 0-15.

So:

```
plot 160,100,0
```

draws a dot at 160,100 with colour zero (black)

and:

```
plot 50,50,6
```

plots a blue dot at coordinates 50,50.

Here's a small example for you to type into AMOS Basic. Hit ESC to return to the editor window, and enter the following listing:

```
do
  Rem Rnd generates a random number from 0 to the value between
  Rem the brackets. So Rnd(1) produces a 0 or a 1
  Plot Rnd(319),Rnd(199),Rnd(15)
loop
```

When you run this program, the editor window will completely disappear, and the display will fill with hundreds of multi-coloured dots. When you get bored, you can exit from the program by just holding down both the Cntrl and the C keys.

At first glance, 16 colours may seem pretty limiting. But each colour index can actually be assigned to one of 4096 individual shades. The 16 indexes therefore represent a small selection from a vast array of possible colours. It's entirely up to you which colour will be displayed by a particular index number. If you're feeling silly, you can even set each index to exactly the same shade. AMOS doesn't care a jot! We can change the shades used of our colour indexes at any time, with the AMOS Basic COLOUR command:

```
COLOUR index,shade
```

index is the number of the colour we wish to set up. Allowable values range from 0 to 31, but the standard screen can only handle 16 colours at a time. I'll be showing you how we can extend this limit later.

shade defines the precise shade of the colour to be assigned to our index. If we want, we can set this number directly to a value from 0 to 4095. Generally however, we use a slightly more complicated format which allows us to mix the shade out of a number of simple components. The shade is entered using the following system:

```
$RGB
```

Each of the letters RGB represents a single digit, and controls the amounts of red, green or blue in the final colour. The higher the value, the brighter and deeper the hue. Some examples are:

$000	RED=0 GREEN=0 BLUE=0	Pitch black
$555	RED=5 GREEN=5 BLUE=5	Dark grey
$900	RED=9 GREEN=0 BLUE=0	Red
$090	RED=0 GREEN=9 BLUE=0	Green
$009	RED=0 GREEN=0 BLUE=9	Blue

Whenever we run one of our AMOS Basic programs, the screen is automatically filled completely with a block of colour 1. Normally, this will be set to orange, but we can change it to flip the display through a Kaleidoscope of colours. Let's try this out. Enter direct mode and type the line:

```
colour 1,$555
```

The entire screen will suddenly turn dark grey! Here are some more values for you to display:

```
colour 1,$900
colour 1,$090
colour 1,$009
colour 1,$990
```

You don't have to continually retype the `colour 1` bit by the way. Just press function key F1 to bring up an exact copy of your previous line, and edit this using the standard cursor keys.

As you may have noticed, the previous colours are all pretty dark. The reason for this is that I've only been using about half the available brightness levels. Each component can really be assigned to one of 16 different intensities. Unfortunately this poses a problem though, as we can't represent 16 values using a single digit. After all, there are only 10 digits available (0 to 9).

We can avoid this dilemma by using hexadecimal numbers. These provide us with a total of 16 digits at once. In order to distinguish them from normal numbers, AMOS expects us to place a $ (dollar) sign before each hexadecimal value. The first 10 digits are identical to the normal decimal system. That's why my previous examples seemed to work out in practice. However, the digits from 10 to 15 are represented by the letters A, B, C, D, E and F. Here's a full list of the 16 hexadecimal digits.

Hexadecimal digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Normal number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now for some examples of these hexadecimal numbers in our colour assignments:

Colour	$FFF	Pure white
Colour	$F00	Bright red
Colour	$0F0	Bright green
Colour	$00F	Bright blue
Colour	$FF0	Yellow

❑ HINT: If you get hopelessly confused during these experiments, type `DEFAULT` from direct mode to get straight back to your original colour settings. Also note that the current colours used by any index can be displayed on the screen using a separate COLOUR function like so:

```
print hex$(colour(1))
```

This returns the colour values assigned to index 1. The HEX$ function just converts the number into hexadecimal so we can check out its individual colour components. With practice you'll be able to visualise a colour straight from the relative intensity values. It's really quite easy once you get the knack.

Currently, I've only changed the shade of colour number 1, but we can obviously define any of the 16 available indexes we like with these functions. Beware of colour number 3 though! AMOS Basic automatically flashes this colour through several different shades whenever it's displayed on the screen. So if we make any changes to this index, they'll be immediately reset by the AMOS system.

```
plot 160,100,3
```

This flashing effect may be occasionally useful, but usually it's just a pain in the neck (also the eyeballs)! It's great for Masochists though! Just draw something in colour 3 and stare at the screen for half and hour. It's guaranteed to give you a splitting headache every time. For the rest of us, it's refreshing to note that it can be turned off completely using a special FLASH OFF command.

```
flash off: plot 100,100,3
```

After we've used FLASH OFF, the colour assigned to index 3 will be totally unpredictable. So it's a good idea to set it straight to a new colour before proceeding.

At the start of our program, it's often necessary to set a whole list of colours at once. Since it would be terribly laborious to have to load each value using a separate COLOUR instruction, AMOS includes a neat PALETTE command which allows us to assign all our colours in a single line.

```
PALETTE c0,c1,c2,c3,c4...
```

loads c0 into colour zero, c1 into colour 1 etc. The number of colour assignments can be anything up to a maximum of 32, but there's no need to set all the colours in a particular screen. We can actually set as few, or as many, colours as we like.

Examples:

```
palette 0
```

Sets colour zero to black.

```
palette 0,$111,$222,$333,$444,$555,$666,$777,
$888,$999,$AAA,$BBB,$CCC,$DDD,$EEE,$FFF
```

Defines a brand new colour palette for the current screen, using shades of grey. Although the example is shown here on two lines, you should enter the entire command in a single line. Just keep typing, and hit RETURN after the final $FFF.

1.2.3 General graphics functions

We are now ready to examine some of the general purpose graphics commands. These provide simple building blocks for the creation our various program displays. We've already discovered how the PLOT instruction can be used to plot simple points on the screen. But it's also possible to draw lines, boxes, and bars, with equal ease.

DRAW tx,ty TO bx,by

The DRAW command draws a straight line from coordinates tx,ty to bx,by. It's more or less equivalent to the LINE instructions found in most other versions of Basic.

Examples:

```
draw 0,0 to 160,100
draw 160,0 to 160,100
draw 0,100 to 160,100
```

These instructions draw a white line on the screen. The drawing colour is set with the help of the AMOS Basic INK command. This has the format:

```
INK index
```

where *index* can be any one of the 16 available colour numbers from the standard screen. However, *index* can actually take values from 0 to 63. This allows us to create custom screens with up to 63 possible colours at a time. See Section 1.3 for more information. Here are some additional examples for you to try:

```
ink 0:draw 0,0 to 320,199
ink 4:draw 160,100 to 320,50
ink 3:draw 160,199 to 160,0
```

Finally, here's a complete program which draws random lines on the screen.

```
Do
  Ink Rnd(15)
  Draw Rnd(319),Rnd(199) to Rnd(319),Rnd(199)
Loop
```

DRAW may be simple, but it's extremely useful in practice. It can be harnessed to generate anything from the sights of a gun to the laser beams in an advanced 3D game. AMOS also allows us to draw whole rectangles on the screen in a single operation. It includes two commands for just this purpose.

```
BOX tx,ty TO bx,by
```

draws a hollow box from tx,ty to bx,by. *tx,ty* set the coordinates of the top left hand corner of the box, and *bx,by* the location of the point diagonally opposite. The sides of the box are drawn using the colour we've set with the previous INK command. This defaults to index 2, which is normally white. Enter direct mode, and try the following examples:

```
box 0,0 to 50,50
ink 6:box 0,0 to 319,199
ink 3:box 80,50 to 240,150
```

Similarly, we can draw filled bars using the AMOS BAR instruction:

```
BAR tx,ty TO bx,by
```

The format of this command is identical to the hollow version. Don't forget to select the colour of the bar with INK before use. Otherwise,all your bars will be white!

Examples:

```
bar 0,0 to 50,50
ink 6:bar 0,0 to 319,199
ink 3:bar 80,50 to 240,150
```

The important thing about the BOX and BAR commands, is that they are very very fast – almost instantaneous in fact! It's therefore quite possible to use them for animation effects in an actual AMOS game.

Up until now, I've only scratched the surface cf AMOS Basic's graphical abilities. But if you've enjoyed this taster, you may wish to look up the following commands in the AMOS Manual. It shouldn't be TOO painful!

CIRCLE x,y,r	Draws a hollow circle
ELLIPSE x,y,r1,r2	Draws a hollow ellipse (egg shape)
POLYLINE x1,y1 TO x2,y2..	Displays a series of connected lines

POLYGON x1,y1 TO x2,y2	Generates an irregular filled shape
PAINT x,y,mode	Fills any hollow screen area with colour. Similar to FILL in other versions of Basic.
SET PATTERN pattern	Chooses one of 34 fill patterns forthe BAR, PAINT, and POLYGON commands
ZOOM	Magnifies or reduces a selected section of our graphics. Great fun!
INK col[,paper][,border]	Has a couple of optional extras which you may find useful!
SHIFT UP/SHIFT DOWN	Rotates the selected colour indexes through a sequence of shades.
FLASH index,s$	Generates a FLASHING colour sequence.
FADE speed[,colour list]	Changes one set of colours smoothly to another.
RAINBOW/SET RAINBOW	Creates an attractive rainbow effect.

Experiment and enjoy!

1.2.4 Clearing the screen

Now for a quick bit of house-keeping. I know chores are boring, but even AMOS Basic has to clean up the mess occasionally. So we'll clearly need a fast way of erasing the current screen contents ready for our next masterpiece. We could use a BAR command such as:

```
Ink COLOUR.INDEX:Bar 0,0 To 319,199
```

However, AMOS also includes a built-in Clear Screen command called CLS. This has several different forms, but the easiest is simply:

```
CLS colour_index
```

Examples:

```
cls 0
```

fills the screen with colour zero.

```
cls 3
```

creates a horrible flashing screen. Turn it off! Turn it off! (Cls 0!)

```
cls 2
```

White-out! CLS is fast, efficient, and wipes out all known graphics. Dead!

1.2.5 Displaying Text

That's enough graphics for the time being. Let's progress to some of AMOS Basic's powerful TEXT commands.

At the start of our Basic programs, a small flashing line _ always appears at the top left hand corner of the screen. If you're curious, you may have wondered what it was for. If you're not curious, you may have been desperately trying to get rid of the damn thing! Well, the explanation is simple enough. It's called the text cursor, and it indicates the exact position of the next character which will be displayed on the screen with PRINT. Enter direct mode with ESC, and type the following line at the AMOS> prompt:

```
Print "Hi"
```

See how the cursor moved straight to the next line of the display. Now try:

```
print "Hi";
```

The semi colon ; tells AMOS to leave the cursor just after the last character in the PRINT statement. So the text cursor will be neatly positioned after the "i".

The cursor can be moved directly to any position on the current screen with a LOCATE command.

LOCATE cx,cy

cx and *cy* are NOT normal screen coordinates, I'm afraid. That would be far too obvious! Unlike our familiar screen coordinates which are measured in units of a single screen point, these text coordinates use units of an individual character. The screen is divided into a rectangular grid, with each box just large enough to hold a single character. Have a look at Figure 1.2. It shows how the letters HI could be positioned at text coordinates 4,2. We could display this text on the screen using a line like:

```
locate 4,2:? "HI"
```

? is just a shorthand form of the PRINT command by the way. It's automatically expanded to PRINT by AMOS Basic, saving us loads of typing.

Now try using LOCATE from direct mode. As you change the cursor position, the cursor bar _ moves as well, so it's easy to see precisely where the next character will be displayed on the screen. After you've familiarised yourself with this system, you may feel the need to remove the cursor from the screen. This can be accomplished with:

```
curs off
```

Ah! That's much better!

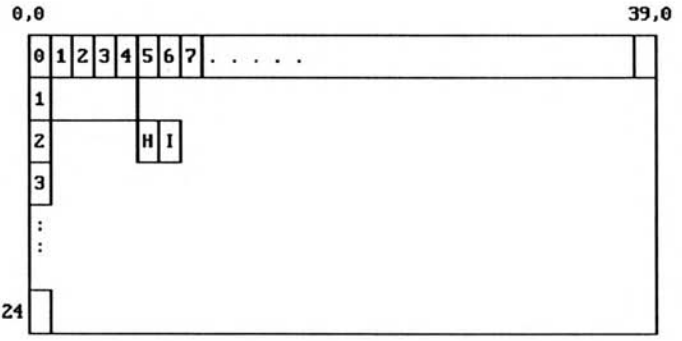

Figure 1.2: Text coordinates

1.2.6 Setting the text colours

Up until now, all our text has been drawn in white ink against an orange background. These colours can be freely changed from within our AMOS program. Type the line:

```
cls 0: print "Hi"
```

Note how the text is displayed against a *paper* of colour 1, regardless of the current screen colours. The background colour can be set using a simple PAPER command.

```
PAPER index
```

Here are some more examples of these text effects:

```
paper 5: print "yellow"
paper 6: print "blue"
paper 10:print "brown"
```

Even the colour of a blank space is affected!

```
paper 3:print "   "
```

generates a flashing bar.

We're not of course, limited to changing the background colour. We can also choose a new colour for our letters as well. The relevant AMOS command is:

```
PEN index
```

For a demonstration, type in the following examples from direct mode:

```
pen 0:paper 1:print "black on white"
pen 3:paper 0:print "flash!"
pen 0:paper 0:print "invisible ink"
```

Now for a larger example which demonstrates the PEN, PAPER and LOCATE commands in action. You should to enter this straight into the AMOS editor window, and run the program by pressing F1.

```
Do
Rem choose random pen and paper colours
Pen Rnd(15) : Paper Rnd(15)
Rem choose a random screen position
Locate Rnd(31),Rnd(24) Print "AMOS text" Loop
```

1.2.7 Converting text coordinates into screen coordinates

With the LOCATE command, we can effortlessly position our text anywhere on the display. Well practically anywhere. Because of the way text coordinates work, we are only able to position our messages in units of a single character. If we want to combine text and graphics in the same program, this can occasionally lead to real problems. Supposing for instance, we wanted to enclose our message with a hollow rectangular box. This would allow us to create a simple "button" effect on the screen.

AMOS provides a useful set of conversion functions which enable us to switch back and forth from text to graphics coordinates at will.

```
x=X GRAPHIC(cx)
```

```
y=Y GRAPHIC(cy)
```

These take the text coordinates *cx* or *cy* and return the appropriate graphic coordinate. Since text coordinates are measured in units of several screen points however, there's no guarantee of an exact match.

Examples:

```
print x graphic(1),y graphic(2)
locate 10,10 : print "BOX"
box x graphic(10),y graphic(10) to x graphic(13),y graphic(11)
```

Naturally, there's also a separate set of functions which convert a text coordinate back into a graphic coordinate.

```
cx=X TEXT(x)
```

```
cy=Y TEXT(y)
```

These convert a graphic coordinate into the nearest equivalent text coordinate. Take the following examples:

```
print x text(8),y text(16)
print x text(10),y text(20)
bar 80,80 to 240,160
locate x text(80),y text(80): print "testing!"
```

Notice how all the coordinates have been automatically rounded down to the closest text coordinate. Each character is exactly eight points wide. So screen coordinates from 8 to 15 all return a TEXT coordinate of 1.

1.2.8 Graphic text

If all this seems hopelessly cumbersome, then you'll be pleased to note that there's a second series of text commands especially designed for use with our graphics. This graphic text treats our text as if it was just another piece of graphics such as a box or a bar. There's no text cursor, no LOCATE commands, just a TEXT instruction with the format:

```
TEXT x,y,text$
```

This displays a string of characters in *text$* at SCREEN coordinates *x,y*. Oddly enough, the y coordinate is measured from the BOTTOM of the text rather than the top. If you forget about this and attempt to position your text at 0,0, only the last line or so of your letters will be displayed. This looks very strange!

Examples from direct mode:

```
text 0,10,"hello"
text 50,50,"AMOS"
text 150,100,"users"
text 0,3,"Whoops!"
```

The colour of the text can be adjusted using an extended version of our INK command:

```
INK pen,paper
```

where *pen* and *paper* are the numbers of colour indexes which are to be used for our text and background areas respectively. Here's an example from the editor:

```
Do
  rem set pen and paper colours to a random value
  Ink Rnd(15),Rnd(15)
  Rem Draw text at random positions
  Text Rnd(319),Rnd(199),"Amos Basic"
Loop
```

Graphic text may be pretty, but it does have its limitations. Unlike PRINT, the TEXT instruction can't display variables or numbers on the screen. It's only capable of printing text. We can sidestep this restriction by loading our numeric variables into a string using a standard Basic function called STR$. If you're relatively new to Basic, you may not have encountered STR$ before. It works like this:

t$=STR$(expression)

t$ can be any standard string variable, and *expression* consists of a calculation involving numbers or variables. The STR$ function first works out the results of this expression, and then converts it into a series of characters held in *t$*.

Examples:

```
a$=str$(10): text 100,50,a$
```

```
age=20:b$=str$(age):text 100,100,a$
```

More advanced expressions can be handled by adding a number of separate AMOS strings together. AMOS provides us with a simply method of combining several smaller strings into a single large one, using a simple "+" operation like so:

```
s$="string 1"+"string 2"+"string 3"...etc
```

This loads *string 1 string 2 string 3* into *s$*. We can now use this system in conjunction with the TEXT, to simulate some of the useful formatting abilities of our original PRINT command.

Supposing we wanted to display our name and age on the screen at coordinates 100,100. These might have been previously placed into the variables *NAME$* and *AGE* respectively. We could load these values into a single large string variable with:

```
message$="Hello "+name$+". Your age is"+str$(age)
```

This could then be displayed at 100,100 using the TEXT command:

```
text 100,100,message$
```

The result would be something like:

Hello stephen. Your age is 30

(I'm getting old!)

We could also include all our values directly in the TEXT command with:

```
text 100,100,"Hello "+name$+". Your age is"+str$(age)
```

I know it looks complicated, but the equivalent PRINT command would have been:

```
print "Hello ";name$;". Your age is";age
```

So all we've done is replaced the ";" with "+", and enclosed the numeric variables with STR$(). Easy, eh? It's important to recognise that these examples only show a tiny fraction of what the TEXT command can actually do. In particular, it's capable of displaying text using a vast range of type faces and styles. For an explanation of these commands, and more, see the AMOS Basic manual under the following headings.

Graphic text

GET FONTS

Loads a set of text fonts from the disk. These must have been previously installed on your disk using a simple procedure outlined in the Extras manual supplied with your Amiga.

GET ROM FONTS

Loads the two typefaces built into your Amiga for use by the TEXT command.

SET FONT

Chooses a typeface to be used for the TEXT command.

SET STYLE

Sets bold or italic or underline styles.

Standard text

AMOS also provides dozens of additional text commands. The creator of AMOS Basic (Francois Lionette) really went to town with the AMOS text system, and it shows! Believe it or not, you could even write a complete word processor entirely in AMOS Basic! (If you do, don't forget to send me a copy. I want one! I want one!)

Alas, I don't have space to detail all these instructions, but here's a brief selection. Apologies to Francois if I've missed out any of his particular favourites!

CENTRE A$

Displays a string of characters centred neatly on the screen.

Example:

```
centre "Hi there"
```

CMOVE cx,cy

Moves the cursor *cx* units right, and *cy* units down from its current position.

Example:

```
locate 10,10:cmove 2,2
```

moves the cursor to 12,12

MEMORIZE X/Y REMEMBER X/Y

MEMORIZE saves the current cursor coordinates in memory, and REMEMBER restores them again to their old positions.

CLINE

Erases the current line of text from cursor position to the rightmost character on the screen.

X CURS/Y CURS

Returns the precise coordinates of the text cursor.

CUP/CDOWN/CLEFT/CRIGHT

Positions the cursor one unit in the selected direction.

=AT$(x,y)

Generates at string of characters which magically move the text cursor to *x,y* whenever they are printed on the screen. For example:

```
print "Hi";at$(10,10);"There"
```

1.3 Creating Custom Screens – SCREEN OPEN

Although the previous instructions are undoubtedly useful, they're hardly exciting, are they? I'll now provide you with a detailed explanation of the more advanced features of AMOS Basic. After all, these are the main reason you bought the package in the first place. Suffice it to say, you won't be disappointed!

AMOS Basic lets us display our graphics in a range of different screen modes. These allow us effortlessly to change both the size and the number of colours used by our AMOS Basic screens. What's more, we can actually display up to eight independent *screens* at a time. Each screen has its own set of colours and can be freely positioned anywhere on the TV display using a clever SCREEN DISPLAY command. Screens are created with an amazingly powerful AMOS Basic SCREEN OPEN instruction. I appreciate that this command is pretty difficult for beginners, so here's a quick breakdown of its important features.

1.3.1 An introduction to SCREEN OPEN

```
SCREEN OPEN screen_number,width,height,number_of_colours,size
```

screen_number holds an identification number of the screen we are currently setting up, ranging from 0 to 7. Screen zero is defined automatically whenever we run an AMOS Basic program, but it can be redefined at any time with a simple call to SCREEN OPEN.

width sets the number of points (or pixels) which will be displayed in each horizontal line of the screen. The usual value is 320, but we can increase this up to about 1000 points. If the screen is too wide to fit onto our TV, we'll obviously only see a fraction of the total area. But our AMOS graphic commands will still be able to access the entire screen directly. This ability is tremendously important for our games, as it can be used to generate dozens of ultra-smooth scrolling effects. (See Chapter 7)

height sets the number of horizontal lines in our new screen. It's normally set to 200, but most European TVs or monitors can easily cope with 255 or even 270 line displays.

number_of_colours chooses the maximum number of colours which can displayed on the screen at any one time. AMOS only allows us to use the following colour combinations in our screens:

Number colours: 2 4 8 16 32 64 4096

The last two combinations are special, as there's a maximum of total of 32 different colour indexes. Despite this, INK can be set to any one of 64 possibilities.

Also note that in order to get the full 4096 colours on a single AMOS screen, you'll need to make use of a system called HAM. See the AMOS Manual for the exact procedure. It's quite involved! Alternatively, draw your HAM screens using a utility such as Photon Paint or Deluxe Paint IV and leave the technicalities to the programming gurus. Ok?

size sets the "size" of each point on the screen. There are just two possibilities.

- Lowres is used by the standard program display, and squeezes a maximum of about 340 points on a horizontal line.
- Hires is used for the AMOS editor and the direct mode window. Since each point is half the width of the lowres version, it's able to cram up to about 670 separate points on each screen line. It's ideal for text, as the screen window can now hold 80 or more characters at once.

By the way, there's also a SCREEN CLOSE command which erases a previously created screen from the Amiga's memory. The format is just:

```
SCREEN CLOSE screen_number
```

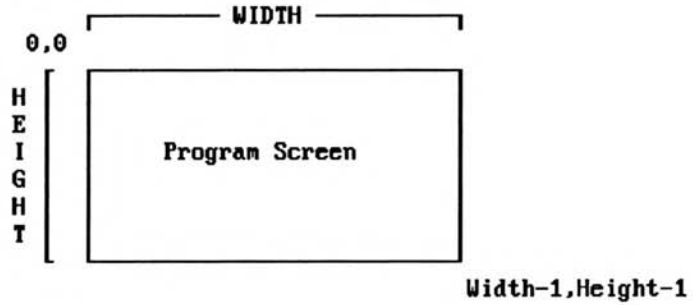

Figure 1.3: The AMOS SCREEN OPEN command

1.3.2 Choosing the screen mode

As you can well imagine, there are literally hundreds of possible screen combinations to choose from. There's no need to panic though. Many of these modes are rarely used in practice, and can be safely ignored. Here's a selection of six of the most useful. Unless you're writing something really advanced, these should be all you need. I've given them numbers just for the purposes of reference, but you don't need them in your AMOS Basic programs.

```
Rem Mode 1: Two colours,standard size,small "pixels"
Rem Great for the text screen of an adventure, as it uses
Rem half the normal memory!
Screen Open 0,640,200,2,hires

Rem Mode 2
Rem Only four colours, but ultra fast screen movements
Rem So it's quite useful when creating mega fast arcade games
Screen Open 0,320,255,4,lowres

Rem Mode 3
Rem A good, all-rounder for arcade games
Screen Open 0,320,255,8,lowres

Rem Mode 4
Rem This is the standard screen display which is created
Rem automatically by AMOS Basic. Despite this, it's perfect for
Rem many games.
Screen Open 0,320,200,16,lowres

Rem Mode 5
Rem Full SCREEN
Rem Note that the screen area is slightly larger than the
Rem visible display
Screen Open 0,360,280,16,lowres
Screen Display 0,112,30,360,280
Rem The screen needs to be positioned manually with a command
Rem Called SCREEN DISPLAY. Feel free to use it and forget it!

Rem Mode 6
Rem 32 colours on the screen at once!
Screen Open 0,320,255,32,lowres
```

Here's a small experimenter program for you to type in.

Example 1.1: Screen Experimenter

```
Rem Screen experimenter
Do
  Rem set up screen
  Rem The next line is pretty standard
  Flash Off : Curs Off : Cls 0
  Rem Enter the screen mode from 1 to 6
  Rem LOCATE in action
  Locate 0,0 : Input " Enter the number of your screen mode?";M
  Rem Jump to the lines at M1,M2,M3,M4,M5,M6
  Rem depending on the screen mode number you selected
  On M Gosub M1,M2,M3,M4,M5,M6
  Rem display colours
  Flash Off : Curs Off : Cls 1 : Paper 0 : Locate 0,0
  Rem This is a neat example of the CENTRE instruction
  Centre "<Colours>"
  Locate 0,4
  SIZE=Screen Width/16
  For C=0 To Screen Colour-1
  Rem Sets the background to colour index C and prints a series of
  Rem coloured spaces
  Paper C : Print Space$(SIZE);
  Next C
  Paper 0 : Locate 0,20 : Centre "<Hit a key to continue>" : Wait Key
Loop Rem Subroutines! See Chapter 2
M1: Rem mode 1
Screen Open 0,640,200,2,Hires
Return
M2: Rem mode 2
Screen Open 0,320,255,4,Lowres
Return
M3: Rem mode 3 Screen Open 0,320,255,8,Lowres
Return
M4: Rem mode 4
Screen Open 0,320,200,16,Lowres
Return
M5: Rem mode 5
Screen Open 0,360,280,16,Lowres
Rem position screen at far left of the display
Screen Display 0,112,30,360,280
Return
M6: Rem mode 6
Screen Open 0,320,255,32,Lowres
Return
```

If you get lost, remember that you can restore the screen back to normal by typing DEFAULT from direct mode. This RESETS screen zero to its original status, and erases any other you've defined as well. Selecting the correct screen for our games obviously involves quite a bit of thought. In particular, the memory consumption varies dramatically depending on the number of colours and the size. I'll be explaining the pros and cons of the various screen modes in Chapter 2.

1.3.3 Creating a screen

Game screens can be created either directly from AMOS Basic or with the help of a separate drawing program such as Deluxe Paint. Once you've drawn your screens, you can load them into AMOS Basic with the LOAD IFF command.

LOAD IFF "filename",screen_number

LOAD IFF loads the picture from *filename* into the selected AMOS screen. If the screen does not exist, it will be automatically created for us by AMOS Basic. IFF incidentally, stands for Interchange File Format. I know it sounds complicated, but it's the standard format used by practically all drawing packages on the Amiga. Thankfully, there's no need to understand how it works in order to use it. I certainly don't!

Anyway, let's load one of these screens into AMOS Basic as an example. Enter the direct mode window and place the AMOS Data disk into the internal drive. Now type:

```
load iff "AMOS_DATA:iff/amospic.iff",0
```

This loads the AMOS title picture straight into screen 0. **Hint:** The size and the number of colours used by the current screen can be found at any time from direct mode:

```
Print Screen Colour
Print Screen Width
Print Screen Height
```

In the case of the AMOS title screen, you'll get the following values:

```
32          No of colours
320         Width
200         Height
```

1.3.4 Multiple screens

A few pages ago, I casually mentioned that AMOS allows us to display several screens at a time. At first glance, this may sound slightly potty, as we've obviously only a single TV screen.

However, since we've total control over the size and position of our AMOS screens, we can divide our display into a number of areas, each with its own individual graphics mode. This lets us make up our game screens out of any combination of the available modes.

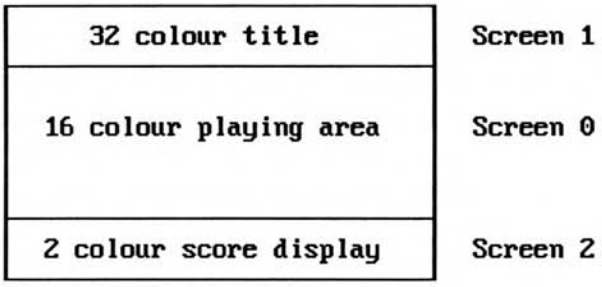

Figure 1.4: Multiple screens

Each area in Figure 1.4 uses a completely separate AMOS screen. Amazing eh? But wait! There's more! Remember that I also said that the shades assigned to the various colour indexes can be totally different for each screen. So the screen in Figure 1.4 can actually display a grand total of 50 colours (32+16+2 = 50) on the screen at once!

We don't have to look far for an example of one of these screens. Take the AMOS direct mode window. This isn't really a window at all – I lied! It's an independent screen which be repositioned over the main program display using the UP and DOWN arrow keys. These types of moving screens are easy in AMOS Basic. I'll be showing you how they can be generated in a short while. Be patient!

Another exciting possibility is to use several full sized screens for the various displays in our games. These can be drawn up during our program's initialisation phases, and then can be flicked into place at a moment's notice. We could, for instance define independent screens for our titles, high score table and game display. We could then flick between the screens to instantly bring up the relevant display at the appropriate point in our program.

A perfect example of this approach can be seen in AMOS Editor window. This holds a complete listing of our current program on the screen, and is entirely separate from the program screen containing our graphics. Yet the two screens can be swapped around almost instantaneously by simply pressing the ESC key from the keyboard.

The ability to generate multiple screens isn't just a useless gimmick. It's an incredibly powerful system which be exploited directly in many games. I'll now explain a few simple techniques which will allow you to use them in your own programs.

1.3.5 Choosing an AMOS screen

The best way of learning about the AMOS screen system is by example. Without any more ado, type the following program into the AMOS editor, and run it with F1.

Example 1.2: Multiple screens

```
Rem Screen Help
Rem Create screen 0 with dimensions 320*200 and 16 colours
Screen Open 0,320,200,16,Lowres
Print "Screen 0"
Print "Press a key to create screen 1"
Wait Key
Rem Create screen 1 of size 640*128, eight colours, small pixels
Screen Open 1,640,128,8,Hires
Print "screen 1"
Print "Press a key to create screen 2"
Wait Key
Rem Create screen 2 of size 320*64 with just four colours
Screen Open 2,320,64,4,Lowres
Print "screen 2"
Print "That's it!"
```

This program creates three screens in quick succession.

- Screen 0 is a 320 by 256 screen with 16 colours.
- Screen 1 is a 640 by 128 screen with 8 colours and hires.
- Screen 2 is 320 by 64 and contains just four colours.

Each screen is exactly half the size of the previous one, so we've the perfect opportunity to see how the screens are actually displayed on our TV.

When we run the program, AMOS displays the current screen number at the bottom of the screen, and then prompts us to hit a key. The previous screen now completely disappears, and the new one appears in its place. But where has the old screen gone? Has it been deleted to

make room for the new display? Nope! It's still there! It's just been covered by the most recent screen.

AMOS automatically displays each screen in the precise order it was created. So the screen at the front is normally the last one we've defined in our program. Like most of AMOS Basic's features though, the display order can be freely changed using the appropriate commands.

SCREEN TO FRONT screen_number

moves the requested screen to the front of the display.

SCREEN TO BACK screen_number

moves a screen behind the display.

The whole process works just like a deck of playing cards. SCREEN TO FRONT moves a card immediately to the top of the pack, whereas SCREEN TO BACK moves it straight to the bottom. By using these commands in succession, we can shuffle the pack into any particular order (see Figure 1.5).

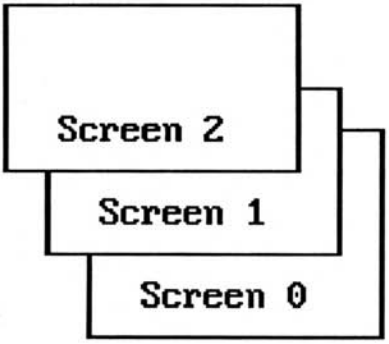

Figure 1.5: Ordering the screen

Let's jump to direct mode, and test this using our previous example. Run the program, and type the following lines from direct mode.

screen to front 1

This moves screen 1 to the front. Screen 2 disappears behind it. Display order:

Top
1
2
0
Bottom

screen to back 1

This moves screen 1 to the back of the pile, and leaves screen 2 at the top.Display order:

Top
2
0
1
Bottom

screen to back 2

Pushes screen 2 to the bottom. Can you work out in advance which screen will appear next? The new display order will be:

Top
0
1
2
Bottom

Have a play with these commands before continuing. The general ideas are fairly straightforward, but a little practice goes a long way.

We'll now examine another feature of Example 1.2. Run the program again, and note how the text is printed on the screen. As you can see, AMOS automatically draws the text on the most recently created screen. The same rule also applies to our graphics as well. They're normally displayed on the last screen we defined in our program. We can select the screen which will be used for our graphics using a simple SCREEN command. There are two forms of this instruction:

```
SCREEN screen_number
```

performs all future drawing operations on the chosen screen. It's important to note that this screen is NOT necessarily the one being displayed on our TV set. It could be completely invisible!

s=SCREEN

This is a function which returns the number of the currently active screen. It's a valuable safeguard which stops us getting lost halfway though our programs.

Let's demonstrate these functions using Example 1.2. Run the program and type the following lines from direct mode.

```
screen 1: locate 0,12: print "Hi there screen 1"
screen 0: locate 0,20: print "Screen 0 is still around"
cls 3
```

This clears screen 2 with colour index 3 (flashing).

```
flash off
screen 2: colour 1,$00F
```

Changes the background colour (index number 1) in screen 2 to bright blue. Notice how the other screens remain completely unaffected!

```
bar 0,0 to 50,50
```

Draws a bar on screen 2.

```
print screen
```

Displays the number of the active screen (2).

In addition to the previous commands, AMOS also provides a couple of instructions which remove a screen from the display list.

SCREEN HIDE screen_number

Hides away the selected screen in memory, and removes it from view.

SCREEN SHOW screen_number

Moves a previously hidden screen to the top of the display.

1.3.6 Positioning an AMOS screen

All our previous screens have been positioned directly on top of each other. But as I explained a while ago, it's possible to move these screens anywhere over the entire display area of our TV. The key to using this feature is provided by the SCREEN DISPLAY command:

SCREEN DISPLAY screen,hx,hy,visible width,visible height

screen is just the number of an AMOS screen we've defined previously with SCREEN OPEN.

hx,hy hold the new position of the screen on the display. *hx* is automatically rounded down to the nearest multiple of 16 by AMOS Basic.

We obviously can't use normal screen coordinates for these values, as they are measured relative to the top left hand corner of the current screen. Instead, we specify the display position using a separate system of hardware coordinates. Hardware coordinates are measured in units of individual screen points, just like the equivalent screen versions. The only difference is that the origin of our coordinates is taken from a reference point to the far left of the screen. The location of this point is dictated by the demands of the Amiga's hardware (Figure 1.6). That's why they're called hardware coordinates.

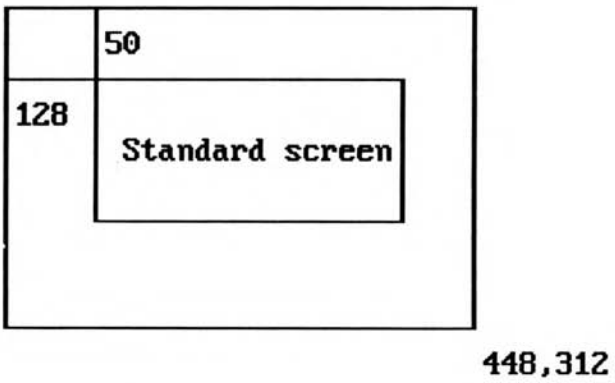

Figure 1.6: Hardware coordinates

The standard screen position is at hardware coordinates 128,50. We can however, move this to anywhere between 112,0 to around 448,312. I'm sorry for the odd movement ranges, but it's not Francois' fault. Alas, they are a direct consequence of the way the Amiga's screen hardware works.

visible width and *visible height* tell AMOS how much of the current screen is to be displayed on our TV or monitor. These settings have NO effect on the physical contents of our screen. They simply change the size of the visible screen region. If we want to use SCREEN DISPLAY to merely position the screen, we can ignore these values completely. But we'll need to include the commas in their original places. For example:

```
SCREEN DISPLAY screen,hx,hy,,
```

Examples (from direct mode):

```
screen display 0,144,100,,
```

moves the screen 16 units right and 50 down. See how I've omitted the width and height values from the instruction.

```
screen display 0,128,0,,
```

positions the screen at the very top of your TV display.

```
screen display 0,128,50,300,128
```

displays a small section of screen zero at coordinates 128,50.

SCREEN DISPLAY is not without its limitations of course. The Amiga's hardware is only capable of displaying certain combinations of screens at once. The biggest restriction is that each horizontal line on our display can only be set to a single screen. So it's not possible to place a number of screens side by side (see Figure 1.7).

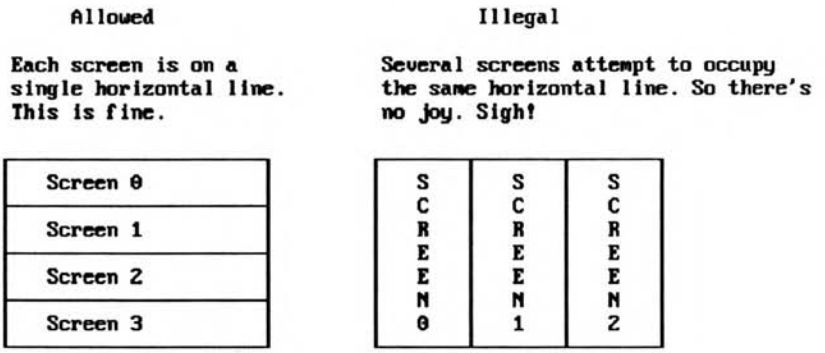

Figure 1.7: Allowable screen combinations

There are also a few limitations on the allowable X coordinates. Some values work fine, and others generate weird results on the screen. Providing we keep within our values range 112 to 400 though, everything will usually be ok. If the display goes crazy, we can just type DEFAULT from direct mode to restore a little sanity.

Just to be on the safe side however, it's worth saving your current program carefully before experimenting. If you're really unlucky, AMOS will crash, leaving you with an indecipherable GURU message and the helpful option to restart your Amiga from scratch. Unless you've saved your AMOS program on the disk, it will be completely lost. Arrgh!

Another problem with SCREEN DISPLAY are those horrible hardware coordinates. Hardware coordinates may seem complicated, but it's easy enough to convert between screen coordinates and hardware coordinates. AMOS provides four special functions which take the pain out of this procedure:

```
hx=X HARD(s,x)
```

```
hy=Y HARD(s,y)
```

These convert an X or Y coordinate on screen *s* into the appropriate hardware version.

x=X SCREEN(s,hx)

y=Y SCREEN(s,hy)

These translate a hardware coordinate into a screen coordinate measured relative to the top left hand corner of screen *s*.

Don't rely on these commands too heavily though. They're not 100 percent accurate, and will occasionally generate impossible results! The good news, is the values are either totally right, or utterly ludicrous. So if the results are within the expected ranges (112-448 for X and 0-312 for Y), we can use them with some confidence. If all else fails, we can always perform the conversion directly in our program. We simply add or subtract the current hardware coordinates of our screen from the appropriate screen coordinates. For example:

hardware coordinate = screen coordinate + screen position

screen coordinate = hardware coordinate – screen position

The screen position holds the hardware coordinates of the screen we've just set with SCREEN DISPLAY.

After the previous explanation, you may feel just a little nervous. In fact, you may decide to avoid SCREEN DISPLAY completely. Surely, it's far too troublesome to use in a real program. Fear not however, for help is at hand. Here's a small AMOS procedure definition which simplifies the system beyond recognition. Feel free to use it directly in your own programs.

```
Procedure MOVE_SCREEN[S,DX,DY]
  Rem Get hardware coordinates of top left corner of screen S
  Rem X coordinate
  SX=X Hard(S,0)
  Rem If SX goes weird, reset to default position
  If SX<112 Then SX=128
  Rem Get Y coordinate
  SY=Y Hard(S,0)
  Rem If Y HARD gives a strange result, reset back to 50
  If SY<0 Then SY=50
  Rem Move screen to new position
  Screen Display S,SX+DX,SY+DY,,
End Proc
```

MOVE_SCREEN adds a useful new instruction to AMOS basic repertoire, and should be added straight to the end of your programs. Don't worry if you're not sure how this works. I'll be explaining AMOS procedures in the next chapter.

MOVE_SCREEN[screen,dx,dy]

Note the SQUARE BRACKETS []. These are vital, as they tell AMOS to interpret the command as a procedure rather than a standard AMOS instruction.

The MOVE_SCREEN procedure moves the selected screen *dx* units right and *dy* units down. Negative values for *dx,dy* just reverse the direction, and indicate a movement of left or up respectively. With the aid of MOVE_SCREEN, we can position our screens with ease. We simply enter the screen number, and specify the distance to be moved in units of a single screen point. When our screens are first created, *dx* and *dy* will initially correspond to our normal screen coordinates. So it's child's play to move all the various screen components into place. What's more, we can forget about hardware coordinates completely! Hurray!

Example:

Supposing we wanted to move the current screen (zero) down 10 units. We could now add the following line to our program:

```
MOVE_SCREEN[0,0,10]
```

When the program is run, the screen will move straight to its new position. But don't forget to include the procedure definition as well, or you'll get an error! Note: MOVE_SCREEN should only be used from INSIDE an AMOS program, as Procedures can't be called from direct mode.

Now think back to the complex screen arrangement I showed you in Figure 1.4. This display could be defined with MOVE_SCREEN like so:

```
Rem Create screens
Rem Defines a 320 by 200 screen with sixteen colours
Rem NSTC users should reduce these values a bit!
Screen Open 0,320,200,16,Lowres
Rem Open screen 1 with 32 colours and dimensions 640 by 50
Screen Open 1,640,50,32,Lowres
Rem open screen 2 with two colours and size 320 by 16
Screen Open 2,320,16,2,Lowres
Rem Moves screen 0 fifty units straight down
Rem So it's at the centre
MOVE_SCREEN[0,0,50]
Rem Since screen one is already at the top,
Rem we can leave it at its original position
Rem move screen 2 directly underneath screen 0
Rem This requires a move of 250 units down
MOVE_SCREEN[2,0,250]
```

We can also use the same routine to move the screen directly inside our program, generating the bouncing screen effects much loved by demo makers.

See the following program fragment, which moves screen zero with the joystick.

```
Rem Move screen zero up by two units
If Jup(1) Then MOVE_SCREEN[0,0,-2]
Rem Move screen zero down by two units
If Jdown(1) Then MOVE_SCREEN[0,0,2]
```

Here's an example of the MOVE_SCREEN procedure in action.

Example 1.3: Positioning an AMOS screen

```
Rem Create a complex screen display out of several modes
Rem I'll be explaining this in a few seconds
Rem Auto View Off
Rem Open screen 0
Screen Open 0,320,200,16,Lowres
Rem move screen 0 fifty units straight down
MOVE_SCREEN[0,0,50]
Curs Off : Flash Off
Rem open screen 1
Screen Open 1,640,50,32,Lowres
Curs Off : Flash Off : Cls 0
Rem Leave it at its original position
Rem open screen 2
Screen Open 2,320,16,2,Lowres
Rem move screen 2 250 units down
MOVE_SCREEN[2,0,250]
```

```
Rem See the next section for an explanation of view
Rem View
Rem Choose screen 0 for text and graphics commands
Screen 0
Rem Display some text
For C=0 To 15
Rem Set colour c to a number from 0 to 15 (Blue)
Colour C,C : Pen C : Locate 0,C : Centre "<Game screen>"
Next C
Rem Choose screen 1
Screen 1
Paper 0
Rem display some random text
For C=0 To 100
Pen Rnd(31) : Paper Rnd(31) : Locate Rnd(30),Rnd(2)
Print "Title screen"
Next C
Rem Choose screen 2
Screen 2
Palette $AAA,$FF0
Curs Off : Locate 30,0 : Print "Score"
Rem see later
Rem Auto View On
Rem main program
Rem Move screen 2 with joystick
Do
Rem I've added a test to these commands to limit screen two to the
Rem visible display area
Rem Y Hard(2,0) holds hardware coordinates of top corner of screen 2
Rem If Joystick is pulled up, then move up screen by two units
If Jup(1) and Y Hard(2,0)>2 Then MOVE_SCREEN[2,0,-2]
Rem Move screen down by two units if joystick pushed down
If Jdown(1) and Y Hard(2,0)<311 Then MOVE_SCREEN[2,0,2]
Loop
Rem MOVE_SCREEN procedure
Rem When you learn more about SCREEN DISPLAY, you'll realise this is
Rem a bit of a compromise. But it's reliable, and easy to use.
Rem So who cares!
Procedure MOVE_SCREEN[S,DX,DY]
  Rem Get hardware coordinates of top left corner of screen S
  Rem X coordinate
  SX=X Hard(S,0)
  Rem If X HARD fails, reset SX to default position
  If SX<112 Then SX=128
  Rem Get Y coordinate
  SY=Y Hard(S,0)
  Rem If strange result, reset to 50
  If SY<0 Then SY=50
  Rem Move screen to new position
  Screen Display S,SX+DX,SY+DY,,
End Proc
```

One slight snag with this system is that the screens are displayed immediately after they are created. This means that the user can see the screens as they are being repositioned on the display. We can side-step this problem using the following commands from AMOS Basic.

AUTO VIEW OFF

Turns off the automatic screen display so any SCREEN OPEN or SCREEN DISPLAY commands will have no apparent effect

VIEW

Displays a screen at its new position (or size).

AUTO VIEW ON

Restarts the automatic screen updates. Any further changes will be reflected immediately on the screen. AUTO VIEW OFF stops AMOS from automatically displaying our screens while they are being initialised. Any changes we make to the display, will be completely ignored.

Once we've created our screens and moved them into position, we can display them on our TV using:

VIEW

The automatic display system can be restarted at any time with AUTO BACK ON. However, this will only affect any FUTURE screen movements. If we want to display the changes since the last AUTO VIEW OFF command, we'll need to call a VIEW instruction as well. I've already included these instructions in Example 1.3. Just remove the Rems from AUTOVIEW and VIEW to see the effect. It's nothing dramatic. It just looks neater.

To round off this section, here's a list of additional SCREEN commands for you to check out in the AMOS Basic manual.

SCREEN OFFSET s,x,y

This is used in conjunction with super large screens. It displays the section of screen *s* starting at coordinates *x,y*. See the section on HARDWARE SCROLLING in Chapter 7.

SCREEN CLONE s

Generates a second version of screen *s* which can be repositioned at resized at will. Weird but occasionally useful.

DUAL PLAYFIELD screen1,screen2

Overlays two complete screens on top of each other.

GET PALETTE(s)

Loads the colours used by screen *s* into the current screen.

DEFAULT PALETTE

Creates the colour palette which will be used for all future AMOS screens.

s=SCIN(hx,hy)

Returns the first screen number at hardware coordinates *hx,hy*.

APPEAR source TO destination,effect

Fades between two screens using a variety of special effects.

1.4 Bobs and Sprites

So much for screens. We'll now look into the problem of generating the small, fast moving objects we'll need in our games. AMOS Basic provides us with two sorts of objects, each with its own unique advantages and drawbacks.

Blitter objects (Bobs) are fast and easy to use, but they can only be displayed on screens with 64 or fewer colours. What's more, as the number of colours in the screen increases, the movement effects slow down dramatically! They're also memory hogs. Despite this, they're still perfect for many games, as their inherent simplicity more than makes up for ANY disadvantages.

Sprites are equally fast, and totally independent of the current screen. They use a fraction of the memory required by Bobs, but they're complicated and unpleasant to use. So we can't win! Hah!

Each object is assigned to a single image from the AMOS sprite bank and can be moved around using a range of simple Basic instructions. We can also access the built-in Amos Animation Language (AMAL) to automatically move our objects through complex preset attack patterns.

If all this baffles you, I'll now quickly run through the general principles of these systems. If you're still stuck, you can find a practical explanation of most of the commands in Chapter 3 (*Arcade games*). You could even check out the relevant sections of the AMOS Basic manual if you're really interested. (Ouch, that HURT!)

1.4.1 The sprite bank

All objects take an image from a memory area known as the sprite bank. (even Blitter objects!). These images can be created in a number of ways. AMOS includes a powerful image editor on the Program disk that provides everything we need to generate full colour images in AMOS Basic. The original editor was called SPRITE_EDITOR.AMOS, but there's also an updated version floating around known as SPRITEX.AMOS. It's supplied free with the AMOS Compiler.

AMOS places no major restrictions on the size of these images, except that the width MUST divide evenly by 16, that is, 16, 32, 48, 64, 80, 96, 112 or 128. The heights can be anything from 1 to 127.

Our images can be drawn in any of the allowable screen modes except 4096 (HAM). But sprites can only make use of images drawn in 4 or 16 colours. Remember, the more colours we use, the slower the movement effects. If you're in doubt, you should draw your objects in the default 16 colour mode. These will work immediately with the standard AMOS screen, and can be assigned to either sprites or Bobs if required.

When we've created our images, they can be saved to disk and loaded straight into our AMOS Basic program using the LOAD command:

```
LOAD "filename.abk"
```

Don't confuse this with the LOAD IFF command. LOAD only handles AMOS memory banks.

filename.abk is the name we've used for our image file. The next time we save our program, all the current sprite images will be saved as well. So we'll now be able to display them instantly from our routines. Neat!

Note that we DON'T have to rely completely on the AMOS sprite editor. There's absolutely nothing stopping us from creating all our images directly from an external program, such as Deluxe Paint. These images can be entered straight into an AMOS image bank using the SPRITE_GRABBER.AMOS routine on the AMOS PROGRAM disk.

1.4.2 Sprite_600

If like me, you've the artistic talent of a dyslexic Gorilla, you may encounter some slight problems in drawing up your sprite images! Thankfully, help is at hand, care of Mandarin (now Europress) Software. In a fit of unbelievable generosity, they kindly included a total of 600 high quality images on the AMOS Extras disk.

These images are divided into several categories such as Aliens, Space, Ground, Fantasy, etc. The disk also includes a clever little demonstration program called SPRITE_600.AMOS that provides us with a simple way of to display all 600 images on the screen. There's a mouth watering selection! We can also get a listing of all the image files from direct mode:

Place the AMOS EXTRAS disk into any drive and enter direct mode with ESC. Now type the following lines:

```
dir$="Extras:Sprites_600/"
dir
```

This brings up a list of all the various categories. We can then display the actual image files using a line like:

```
dir$="Extras:Sprite_600/Aliens/"
dir
```

Finally, we can load the images from a particular file with a command such as:

```
load "Extras:Sprite_600/Aliens/Alien1.abk"
```

These images are drawn in 16 colour (lowres) mode. So they can be used straight with the standard AMOS screen.

1.4.3 Blitter objects (Bobs)

Once we've loaded our images, we'll need some way of displaying them. The easiest approach is to make use of Blitter objects or Bobs. Bobs are stored directly as part of the current screen using the infamous Blitter chip, and can be displayed in any of the Amiga's screen modes (except HAM). They are superb for generating the movement effects required in arcade games, as they are fast and colourful.

On the down side though, Bobs do require fair amounts of memory in order to work. If you've got an expanded machine that won't be a problem. But if you're restricted to the standard 512k, then you could occasionally suffer from lack of memory. Don't despair, as most of the demo programs on the AMOS Data disks use Bobs. So you'll still be able to write some pretty large games with this system. For the time being, add the following lines to the start of your programs:

```
CLOSE WORKBENCH
CLOSE EDITOR
```

This will free up the maximum possible memory for your programs.

Controlling a Blitter object is simple. There's a built-in BOB command which does the whole thing for us:

```
BOB b,x,y,image
```

The BOB command positions our Bob on the screen and assigns an image to it from the sprite bank. The image can be changed at any time, generating a variety of attractive animation effects.

b holds an identification number ranging from 0 to 63. It will be used to refer to the Bob in all future commands.

x,y specifies the position of the Bob on the screen, using a couple of standard screen coordinates. Note that AMOS places absolutely NO restrictions on these coordinates. If the Bob image lies within the current screen area, AMOS will happily display at the appropriate screen location. Otherwise it will ignore it completely. This allows us to *kill off* a Blitter object by moving it to a ridiculous screen position such as –100,-100.

image chooses the number of an image from the sprite bank which will be displayed by the selected Bob. Allowable image numbers start from 1. We can get an instant read-out of the number of available images from direct mode with:

```
Print Length(1)
```

Note that since the image will be displayed on the existing screen, it needs to be in the current screen mode. In other words, the image should use the same number of colours as the screen. If for instance, we attempt to display an eight colour image on a 16 colour screen, we may get some VERY odd effects. What we won't get, is an error! AMOS sensibly leaves the image selection entirely up to us. This opens the door to a range of useful effects. See the SET BOB command in the AMOS manual for more details.

Anyway, it's time for a quick example. Place the Extras disk into any drive and load some images with:

```
Load "Extras:Sprite_600/aliens/alien1.abk"
```

Next, assign an image to Bob number 1 using a command like:

```
Bob 1,160,100,1
```

We now have a strange looking alien sitting at the centre of our TV screen. The reason it looks so odd is that the colours used by the Bob are different from those used by the current screen. Fortunately, these Bob colours are automatically saved in the sprite bank along with our image data. We can load them into our screen using the GET SPRITE PALETTE command.

Try typing:

```
Flash off:Get Sprite Palette
```

That's much better! See how I've used the FLASH OFF instruction BEFORE loading the colours with GET SPRITE PALETTE. Try reversing the order and see what happens. It's not exactly pleasant is it?

Now for an example which demonstrates how Bobs can be moved on the screen. This should be entered into AMOS Basic and run from the editor window with F1.

```
Rem Load some images from the extras disk
Load "extras:sprite_600/aliens/alien1.abk"
Flash Off : Get Sprite Palette
Cls 0
Do
Rem Move the blitter objects
For X=1 To 320
    Bob 1,X,0,1
    Bob 2,X,50,1
    Bob 3,X,100,1
Next X
Loop
```

Yeuck! What have we done wrong? Well, the AMOS Bob commands require us to use a system known as double buffering. This creates the Blitter objects in an invisible copy of the current screen, and only displays them when they are fully drawn. You don't need to worry about how this works, as the process is completely automatic! All you need to do, is to add a single DOUBLE BUFFER command at the start of your program. AMOS does the rest!

```
Load "Extras:sprite_600/aliens/alien1.abk"
Flash Off : Get Sprite Palette
Rem Activate double buffering
Double Buffer
Cls 0
Rem Move the Bobs across  the screen
Do
For X=1 To 320
    Bob 1,X,0,1
    Bob 2,X,50,1
    Bob 3,X,100,1
Next X
Loop
```

The flaw with DOUBLE BUFFER of course, is that it also doubles the amount of memory used by the current screen. Sigh!

If you're still not satisfied, you'll be delighted to discover that the effect can be improved even further with the UPDATE command. This performs all the object movements at a single point in your program. I'll be discussing it in detail in Chapter 3. Here's a taster of the types of effects which can be achieved.

```
Load "extras:sprite_600/aliens/alien1.abk"
Rem Boring initialisation stuff
Flash Off : Get Sprite Palette
Double Buffer
Cls 0
Rem Stop the Bobs from being redisplayed when they are moved
Update Off
Do
Rem Move blitter objects across the screen
Rem (The STEP 4 bit simply speeds things up a little!)
For X=1 To 320 Step 4
Bob 1,X,0,1
Bob 2,X,50,1 Bob 3,X,100,1
Rem Redraw all the Bobs at their new positions
Update : Wait Vbl
Next X
Loop
```

Before we proceed, it's worth noting an extra little feature of the Bob command. Once we've defined our object and positioned it on the screen, we can optionally omit any of the values X, Y or I from any future Bob commands. But we still need to include all the commas in their original places. So the following instructions are all perfectly acceptable:

```
Bob 1,100,100,1
```

Defines our initial screen coordinates and chooses an image for it.

```
Bob 1,,10,1
```

Moves the Bob to 100,10. The X coordinate remains unchanged.

```
Bob 1,100,160,
```

Moves the Bob to 100,160 using the original image number.

The AMOS BOB command really is an amazing instruction. So it's worth playing around with it for a while before continuing. I'll be using it a LOT in my forthcoming examples! If you fancy a spot of further reading, I'll now list some more of the AMOS Bob commands for you to check out from the manual

x=X BOB(b)

y=Y BOB(b)

Returns the current X and Y coordinates of Bob *b*.

GET BOB image,tx,ty TO bx,by

Loads a Bob image from the screen into the sprite bank.

LIMIT BOB tx,ty TO bx,by

Restricts Bob movements to a small area of the screen.

PASTE BOB x,y,i

Pastes a permanent copy of image number *i* at screen coordinates *x,y*. Great for fast drawing effects.

BOB OFF b

Removes Bob *b* from the screen and deletes it from memory.

BOB CLEAR/BOB DRAW/BOB UPDATE

Control the drawing system used by the Bob. See Chapter 3.

SET BOB

Changes the way the colours of our Bob will be displayed on the screen. It's pretty technical stuff, but it's well worth a look.

BOB COL (Bob number,first to last)

Checks for a collision between two or more blitter objects.

1.4.4 Sprites

Sprites are similar to Bobs except that they are generated directly by the Amiga's hardware, and are totally independent of the Amiga's screen modes. So they can be used in any graphic mode we like, without the need to worry about the DOUBLE BUFFER command.

Unfortunately they are quite tricky to control in practice, and there are dozens of extra restrictions to their use. So most AMOS programmers tend to ignore them and concentrate on the simpler Blitter objects instead. If you are a complete beginner, feel free to follow their example and jump immediately to the next section. You certainly don't need to understand any of this to write cracking good games!

The Amiga's hardware provides the programmer with eight special objects which can be automatically overlaid on the current screen. Each of these hardware sprites is exactly 16 points wide, and can be anything up to 255 units tall. It can also display a maximum of four colours.

As they stand, hardware sprites aren't exactly earth-shattering are they? But AMOS Basic takes these simple components and cleverly combines them to display up to 48 computed sprites on the screen at once. Even the colour restriction is removed. Computed sprites can display up to 16 colours! This is a tremendous achievement, and required some fancy programming on the part of Francois!

The complication arises because each computed sprite is actually composed of several hardware sprites. That poses a definite limit on the number and type of computed sprites which can be handled by the AMOS system. In order to display the computed sprites on the screen, AMOS is forced to go through a complicated juggling act. So it's not surprising that an object falls through its fingers occasionally.

Well, that's the general theory. Thankfully, the practice is slightly easier.

Sprites can be displayed using the AMOS Basic SPRITE instruction.

`SPRITE number,hx,hy,image`

number holds the number of the sprite you wish to display. Sprite numbers from 0-7 access the hardware sprites, and those from 8 to 63 use computed sprites. Since hardware sprites are limited to 16 units wide, we'll usually want to use computed sprites. Then again, individual hardware sprites don't suffer from the movement restrictions posed by the more advanced computed sprites. There's therefore a genuine trade-off between the two systems.

hx and *hy* set the position of the sprite on the screen. Alas, these coordinates are NOT the same as standard screen coordinates used by the Blitter objects. These are the same hardware coordinates needed by the SCREEN DISPLAY command. They are also used to hold the current position of the mouse pointer. This isn't a coincidence, as the mouse pointer is actually displayed using hardware sprite number zero! In fact, if we want to use the maximum number of computed sprites, we'll first need to remove the mouse pointer from the screen with HIDE. Otherwise, there'll be only seven hardware sprites available for the AMOS system.

image is the number of an image taken from the sprite bank. The same images are used for both sprites and Bobs. We can therefore easily combine sprites and Bobs in the same program. Only 4 or 16 colour images can be used however. As a compensation, sprites can be displayed equally well in ANY of the Amiga's graphics modes, including 4096 (HAM). So we can easily generate moving sprite images against a 64 or 4096 colour background!

Also note that the maximum allowable width for a four colour image is 128, but this is only available if we've previously removed the mouse pointer with HIDE. Similarly, 16 colour images can be anything up to 64 units wide (or 48 with the mouse pointer).

Let's fire up AMOS Basic and display one of these sprites on the screen. Place the Extras disk into any drive, and type the following lines:

```
dir$="EXTRAS:Sprite_600/water/
```

Sets the current directory to something fishy!

```
load "shark.abk"
```

Loads up an image file. Surprise surprise! It's got a shark in it!

```
sprite 8,228,150,1
```

Positions computed sprite 8 at hardware coordinates 228,150 (100,100 in screen coords) and assigns our shark to it.

At the moment the colours look weird! So we'll need to grab the image colours from the sprite bank using the GET SPRITE PALETTE command as before.

```
flash off:get sprite palette
cls 0
```

Blanks out the background so we can see our shark.

```
sprite 9,228,180,2
```

Another day, another shark!

```
sprite 10,228,210,3
```

As you can see, we've no problems at all in positioning our sprites below each other. But let's try displaying a couple side by side.

```
sprite 10,328,210,3
```

It's ok here.

```
sprite 10,328,180,3
```

When we attempt to position sprite number 10 directly alongside one of its neighbours, it vanishes from the screen completely!

The reasoning behind this is fairly involved. Each of the eight hardware sprites is actually being reused several times to generate our three computed sprites on the screen. But the Amiga's hardware just can't display the same sprite more than once on each horizontal screen line. So if we try to exceed this number,and place more than eight sprites on a horizontal line, AMOS is forced to give up in disgust.

Rather than cause mayhem on the screen by attempting the impossible, AMOS sensibly ignores our sprite command completely, and continues executing our Basic program as if nothing had happened.

Of course, our shark actually represents a pretty extreme case of this effect, as it's 48 units wide and uses the maximum 16 colours. If we were prepared to reduce the number of colours to four, we could easily double the number of sharks on the screen. At its best, AMOS can display up to eight 16 by 16 (four colour) sprites on each horizontal line. That's not bad!

The practical upshot of all this is that although we can have as many as 48 computed sprites on the screen at once, we can't display our objects in any order. This poses real problems for our games. In an arcade game, our aliens could be moving in quite complicated attack patterns. So it would be extremely hard to work out in advance exactly which combinations of sprites would be displayed alongside each other. If we made a mistake, the aliens would totally overload the sprite system, and wink on and off at irregular intervals as AMOS struggled to display them all on the screen.

As always, we need to experiment carefully to produce the best results. If we need just a few objects, sprites are often ideal. This is especially true in games which require lots of colours on the screen, as we can move our sprites far faster than the equivalent Blitter objects. But sprites are not really suitable for general purpose programming.

One useful command to watch out for is SET SPRITE BUFFER. This increases the amount of memory allocated to the AMOS sprite system. As a default each hardware sprite can be up 128 units tall, but it's a good idea to increase this to the maximum of 256 at the start of our programs. This will allow us to cram several additional sprites onto the screen at a time.

Use a command like:

```
SET SPRITE BUFFER 256
```

Here's an example for you to play around with:

```
Hide Load "EXTRAS:Sprite_600/water/shark.abk"
Flash off:Get Sprite palette
Rem Set Sprite buffer 256
Rem display then computed sprites near the centre of the screen
For S=0 To 10
Sprite S+8,150+128,S*20+50,1
Next S
Wait Vbl
```

If you run this program as it stands, part of a shark will be missing from the bottom of the screen. Remove the Rem statement before the Set Sprite Buffer command for an extra fish!

The WAIT VBL command incidentally, just synchronises the sprite movement with the screen display. OK?

If you're still struggling with hardware coordinates, don't forget that they can be converted straight into the normal screen versions using the functions:

sx=X SCREEN(hx)

sy=Y SCREEN(hy)

These take a set of hardware coordinates in *hx,hy* and return the equivalent screen versions in *sx,sy*.

We can also convert our screen coordinates back into hardware coordinates, with the X HARD and Y HARD functions like so:

hx=X HARD(sx)

hy=Y HARD(sy)

Sprites may be a little fiddly, but they aren't totally useless. Providing we treat them with a little caution, they are still capable of generating some pretty neat effects. They'll therefore rate a small mention in the forthcoming chapter on Arcade Games.

Example:

```
Load "extras:sprite_600/aliens/alien1.abk"
Flash Off : Get Sprite Palette
Cls 0
Rem The UPDATE system can be used as in the previous example
Rem Use Update Off
Do
Rem Move the sprites across the screen
For X=1 To 320
Sprite 8,X Hard(X),Y Hard(0),1
Sprite 9,X Hard(X),Y Hard(50),1
Sprite 10,X Hard(X),Y Hard(100),1
Rem Add some smoothness with Update : Wait Vbl
Next X
Loop
```

A further explanation of the pros and cons of the AMOS SPRITE commands can be found in the AMOS manual. If you've followed the previous discussion, you should find it fairly straightforward. Also see the following instructions:

hx=X SPRITE(s)
hy=Y SPRITE(s)

Returns the hardware coordinates of sprite *s*.

SPRITE OFF [n]

Removes one or more sprites from the screen.

SPRITE UPDATE

Synchronises the sprite movements with an animated screen.

SPRITE COL

SPRITE BOB COL

BOB SPRITE COL

Collision detection between sprites and Bobs.

1.5 The AMOS Animation Language (AMAL)

AMOS doesn't just provide us the ability to display our objects. It also includes an advanced animation system which allows us to whizz them across the screen at high speed.

The AMOS Animation Language (AMAL for short) affords us an easy way of generating the fast and smooth movement effects which are the hallmark of the modern arcade game. It works by taking an object and animating it using a series of simple (Basic like) instructions. These are usually stored in a standard AMOS string variable, but we can also load them into a memory bank using a powerful AMAL Editor program. In case you've never used it, it's in the file AMAL_Editor.AMOS on the AMOS Extras disk.

AMAL programs are normally executed automatically by the AMOS system, 50 times a second. So once we've defined our animation routines, we can basically forget about them. All our objects will move and animate independently, even if our program is doing something completely different. It's almost as if they've got a life of their own!

Although AMAL supports a total of 28 separate commands, we can usually get by with a mere seven or so. Here's the list:

Anim	Animates an object through a number of images.
Move	Moves an object on the screen in a straight line.
Let	Assigns a value to an AMAL variable.
Jump	Like GOTO. It Jumps to a label defined elsewhere in our AMAL program.
If	Similar to the AMOS If command
For..Next	Loops, just like those in AMOS Basic.
Pause	Another version of the mysterious Wait Vbl command

I'll be discussing these commands in Chapter 3. As you read, you'll discover how to create anything from a simple joystick handler to an entire alien fleet moving rapidly across the screen. For the moment though, I'll limit myself to a brief explanation of the general principles.

1.5.1 Variables and registers

AMOS Basic is capable of executing up to 64 separate AMAL programs simultaneously. Each program controls the movement or appearance of a single object on the screen. Like Basic, AMAL allows us to control our objects using a series of simple variables. These AMAL registers can be set to any value from –32768 to +32767. They are loaded with a number using the AMAL Let instruction.

Let register=expression

register is the name of the AMAL register we wish to change, and *expression* can consist of either a number or a simple calculation.

Some registers are special:

A — Holds the image number currently assigned to our object. If we change it, using Let we can animate our object through several different images. For example: `Let A=10` assigns image number 10 to our selected object. Incidentally this only works for Bobs and sprites .

X,Y — Stores the coordinates of our object in memory. Whenever we load a new value into these variables, our object moves accordingly on the screen. For example: `Let X=100; Let Y=100;` moves our object to coordinates 100,100.

AMAL also provides us with 10 local registers to contain temporary information such as counter in a For..Next loop, or the size of our movement patterns. Local variables are given standard names in the format:

R<Digit>

where <*Digit*> can be any single number from 0 to 9. Allowable names range from R0 to R9.

Each AMAL program has its own separate set of local variables. So a command like `Let R0=10` only applies to the current AMAL program! In another AMAL program, the value could be completely different!

Alongside the local registers there's also a set of 26 global registers (RA to RZ). These keep the same value throughout the entire AMAL system. If we want to access these registers from an AMOS Basic program, we'll need to use the following commands.

v=AMREG(program number,r)

Reads the contents of the local register assigned to the appropriate AMAL program. *r* is a number from 0 to 9 which indicates which register we wish to read.

v=AMREG(r)

Reads a global register. In this case, *r* lies in the range 0 to 25. RA is represented by a zero, RB by a 1, and so on up to RZ (26). Here's a simple trick which allows us to convert the register name into the appropriate number. Just enter the letter after the R into the ASC function like so:

```
V=Amreg(Asc("F")-Asc("A"))
```

This loads the contents of global register RF into the AMOS variable *V*.

There's also a second version of these routines which let us load a register from an AMOS variable. The format is simply:

AMREG(r)=v

AMREG(program,r)=v

So as an example:

```
amreg(0,5)=10
```

stores 10 in register R5 assigned to program 0.

1.5.2 Defining an AMAL program

Before we define an AMAL program, we first need to assign it an object. This is accomplished using the AMOS Basic CHANNEL command:

CHANNEL channel_number TO type object_number

channel_number is the number of the AMAL program we are intending to define, ranging from 0 to 63. Numbers from 0 to 15 are moved automatically by AMOS Basic, 50 times a second. The rest can only be used from the SYNCHRO system I'll be showing you in Chapter 3.

type is one of six AMOS Basic keywords which chooses the type of object to be animated. Here's a list of the three most useful options.

Bob Moves or Animates a Blitter object. This is easily the most common use for the AMAL system.

sprite Controls a sprite.

screen display Moves a complete AMOS SCREEN, creating those bouncing screen windows you may have seen in certain demos.

The object_number chooses an object to be controlled by the AMAL animation string. This object DOESN'T need to have been defined previously in our program. Providing it's available when our AMAL commands are finally executed, everything will be fine.

Examples:

```
channel 1 To bob 1
```

Assigns AMAL program number 1 to blitter object 1.

```
channel 2 To bob 10
```

Allocates Bob number 10 to channel 2.

```
channel 3 to sprite 8
```

Animates computed sprite 8 using channel 3.

Once we've set up the channel, we can then enter our animation commands into a string, and call up our AMAL program using the AMAL command. The format is simply:

AMAL channel_number,program$

channel_number is the number we used to select our object in the previous channel instruction.

program$ is an AMOS Basic string containing a list of all the AMAL commands we wish to execute. Let's see how this works with a quick example from direct mode:

```
load "Extras:Sprite_600/flight/helecopt.abk"
```

Loads up some images. Note the odd spelling for "helecopt.abk"

```
flash off:get sprite palette:cls 0
```

Initialises the screen.

```
channel 1 to Bob 1
```

Allocates Channel 1 to Bob 1

```
bob 1,160,50,1
```

Displays a helicopter.

```
amal 1,"Anim 0,(1,5)(2,5)(3,5)"
```

Loads an AMAL program into channel 1.

```
amal on
```

Powers up the AMAL system and starts our AMAL program running. Now sit back, and watch AMAL animate those rotor blades! Try moving the direct mode window using the UP or DOWN arrows from the keyboard. The animation continues unaffected! Now jump to the editor with Esc. When you return to direct mode, the helicopter's still there, and it's STILL being animated!

This demonstrates how AMAL programs are COMPLETELY independent of the AMOS system. Now for a larger demo:

```
Rem load up images from the extras disk
Load "Extras:sprite_600/flight/helecopt.abk"
Rem set up screen
Flash Off : Get Sprite Palette : Curs Off : Cls 0
Double Buffer
Rem Initialise AMAL
or H=0 To 4
Rem Use channel H to control Bob H
Channel H To Bob H
Rem position each helicopter 30 units apart
Bob H,300,30*H,1
Rem Assign an AMAL command string to helicopter number H
Amal H,"Anim 0,(1,5)(2,5)(3,5)"
Next H
Rem Start Blades whirring!
Amal On
Rem Move helicopters across the screen using BOB
For X=300 To 1 Step -1
For H=0 To 4
Rem Note the commas. I'm only changing the X coordinate
Rem So the Y coordinate and image number used by the Bob
Rem remains the same.
Rem Remember the Bob command is: Bob b,X,Y,IMAGE.
Rem I've omitted Y and IMAGE, but left the commas.
Bob H,X,,
Next H
Rem Perform a simple bit of synchronisation
Wait Vbl
Next X
```

These examples made use of the Anim command. Anim allows us to flick our object through a series of simple images stored in the sprite bank.

1.5.3 An introduction to the Anim command

Anim works like this: The first value sets the number of times the sequence will be repeated. A zero in this position means for ever. So the sequence will continue until you switch it off (AMAL OFF). The stuff between the brackets tells AMOS which image is to be shown and how long it should be displayed. The format is:

`Anim count,(image,time)...`

image holds the number of an image from the sprite bank. *time* represents the amount of time the image will be held on the screen measured in units of a 50th of a second. Here's another example of the ANIM command, which generates a neat explosion effect:

```
Load "Extras:sprite_600/space/ship1.abk"
Flash Off : Get Sprite Palette : Curs Off : Cls 0
Double Buffer
Rem define animation string
A$="Anim 1,(21,5)(22,5)(23,5)(24,5)(25,5)(26,5)(27,5)(28,5)(29,5)(30,5)"
Rem set up channels and position object on the screen
For S=0 To 4
Rem Use Bob S for channel S
Channel S To Bob S
Rem position each objects apart
Bob S,S*32,30*S,1
Rem assign explosion effect in A$ to Channel number S
Amal S,A$
Next S
Locate 0,22 : Paper 0 : Centre "Press a key to destroy ships"
Wait Key : Boom
Rem Start explosion!
Amal On
Wait 50
```

Look out for other explosion sequences on the SPRITE_600 files. There are dozens of them to choose from.

It's vital to realise that AMAL only uses the first few letters of each command. These are entered in CAPITALS. Anything else is ignored completely. So only the A of the Anim instruction is important. The nim will be utterly ignored by the AMAL system. It's only included to make the command more readable. If you're feeling silly, you can happily rename it to *Arnold* instead. Your AMAL program will continue as normal!

1.5.4 The AMAL Move command

Another useful AMAL instruction is Move, which allows us to move our objects smoothly across the screen in a simple straight line. It's perfect for the missiles in an arcade game. Like all AMAL commands Move cannot be entered directly. It first needs to be placed into a string and executed using the AMAL command:

`AMAL 1,"M 100,100,50"`

Don't forget to assign a object to this animation using the AMOS Basic Channel command. Otherwise absolutely NOTHING will happen! The general format of the Move command is simply:

`Move horizontal distance,vertical distance,n`

The horizontal and vertical distances enter the total number of units our object is to be moved across the screen.

`Horizontal distance:`

A positive value moves the object to the right and a negative value to the left.

Vertical distance:

Positive is down, Negative is up.

The motion is divided into exactly *N* separate movement steps. Take the following example:

```
Move 150,100,50
```

moves an object 150 units right and 100 units down in 50 individual steps. The size of each movement is: 150/50 =3 units right 100/50 =2 units down

So our object would move steadily towards the bottom right of the screen. Incidentally, there's nothing stopping us from using negative values for these distances as well such as:

```
Move -150,-100,25
```

This moves the object 150 units left and 100 units up in 25 steps.

Example from direct mode:

```
load "extras:sprite_600/aliens/alien3.abk"
```

Load some images.

```
flash off:get sprite palette:cls 0
```

Initialisation stuff!

```
channel 1 to Bob 1
```

Allocate channel 1 to Bob 1.

```
Bob 1,0,0,1:Amal 1,"Move 300,200,100"
```

Assign a movement string to our AMAL program.

```
Amal on
```

Move it! Amal now moves our alien slowly to the corner of the screen.

We can also combine several movement commands in the same AMAL program.

```
Bob 1,0,0,1:Amal 1,"Move 300,200,100;Move -300,-200,100"
```

See how we've separated our commands with a semicolon. We can also use a space as well. But unlike AMOS Basic, we can't separate our AMAL commands with a ":" character. AMAL uses this colon to define labels. It's only a small difference I know, but it can have a dramatic effect on the working of our AMAL programs!

Note that for the best results, the number of movement steps in our command MUST divide evenly into both the horizontal and vertical distances. Suppose we used an instruction like:
`Move 130,80,50`

The sizes of each step would now be:

130/50 =2.3 units right 80/50 =1.6 units down

Since it's impossible to move an object by a fraction of a unit, AMAL would be forced to change the size of the movement steps on the fly. The vertical steps would vary between 2 or 3 right and the horizontal values would be either 1 or 2 units down depending on the size of the step. As you can imagine, the resulting motion would be extremely jerky!

Here's a larger example for you to play around with. It demonstrates how AMAL can be used to create simple attack patterns on the screen. A complete explanation of this system can be found in Chapter 3.

```
Rem Attack Attack
Load "Extras:sprite_600/aliens/alien1.abk"
Flash Off : Get Sprite Palette : Curs Off : Cls 0
Double Buffer
Rem Here's a long AMAL definition split between several lines
Rem I've added each line to the end of the previous one using "+"
a$="MOVE 0,100,50; MOVE 250,0,50; MOVE 0,-100,50; MOVE -250,0,50;"
B$="Loop: "
B$=B$+"Move 0,100,50; Move 250,0,50; Move 0,-100,50; Move -250,0,50;"
B$=B$+"Jump Loop"
Rem Define channels, position alien, and load animation string
For A=0 To 4
Rem Channels 0 to 4 are assigned to the appropriate Bob
Channel A To Bob A
Rem position each alien 32 units apart
Bob A,32,25*A,1
Rem Load the AMAL commands in A$ into the CHANNEL
Rem we've just defined
Rem also try B$
Amal A,A$
Next A
Rem Wait for the disk to stop whirring!
Wait Vbl
Locate 0,11 : Paper 0 : Pen 1 : Centre "AMAL in action!"
Rem Start all AMAL programs moving
Amal On
Rem Do nothing! Let the AMAL program do the work!
Do
Loop
```

This program also includes a demo of the Jump command, which is AMAL's equivalent to the familiar AMOS GOTO instruction. Jump can be used to construct anything from a simple loop to an complex test.

However, it's important to note that unlike AMOS Basic, AMAL only allows us to define up to 26 different labels in a program. Each label followed by a single colon (:) . These are assigned to the CAPITAL letters A through Z.

Since AMAL will ignore anything in lowercase (or small letters), it's common practice to pad out our labels with extra letters in order to make them more readable. We could therefore replace our A in the previous example with *Aloop:* instead. But remember, only the first letter (the A) is important. So we can't define another label such as *Able:* and expect it to work. This will generate an error message such as *Label defined twice*, as it's illegal to define two or more labels starting with the same capital letter.

If you want a quick demonstration of the Jump command in action, replace the *A$* in the `AMAL A,A$` with *B$*. This repeats all our the movements continually on the screen. Notice how I've split the AMAL commands in *B$* between several lines. I'll be using this technique extensively in my future examples.

If you're feeling adventurous, you may want to play around with AMAL some more. I'll be going into this AMAL movement system in great depth in Chapter 3. You can also find an AMAL tutorial in Chapter 14 of the AMOS Basic manual. Now you know the basic ideas, you should find it pretty easy.

Whew! If you've reached this far, you're already well on the way to being an AMOS expert. All you need to do now is write some games! But where do you actually start? That's the purpose of the next chapter, where I'll be explaining how you can harness these instructions to create full-blown AMOS Basic games! Have fun!

2

Think Before you Leap

2.1 Planning a Game

It's a popular misconception that the hardest part of games programming is writing the actual code. If you were to ask a professional programmer about this though, the answer would probably surprise you. Programming is the easy bit, they would say, it's only the initial planning phases that are really complicated. Commercial software houses now employ separate people to make up the initial plans and think up the game ideas. These *systems analysts* don't produce a single line of code, but they are usually paid several times the salary of even the best computer programmer. The reasoning is simple. Talented programmers are fairly commonplace, but original game ideas are unbelievably rare!

The Amiga software scene is littered with games which look wonderful, sound great, and yet bore us to tears after a couple of minute's play. The programming could be brilliant of course, but we never actually see it! We're too busy playing the game! This type of problem is the direct result of sloppy planning. The tragedy though, is that it's often totally preventable. Sometimes the temptation to start programming immediately can be almost irresistible. But it's vital to avoid this temptation at all costs! Any initial mistakes will be propagated through your program like a horde of locusts, and will eat away at it until it is stone dead. After you've invested a great deal of time and energy into a particular program, it can be heartbreaking to discover that the basic idea is completely unworkable.

Writing a game without a plan is as daft as visiting a strange city without bothering to take a map. In both situations you would probably get totally lost before you reached halfway to your intended destination. Planning is one of the most crucial phases of games creation. Good programmers plan each step of their programs carefully before typing a single line into the computer. So when they get down to writing their programs they always know precisely what they are trying to achieve, and have already solved most of the potential problems well in advance.

If you plan your game scrupulously, the programming will seem easy. Once you've mapped out the general route, you can relax and enjoy your journey. Often, a single day's worth of planning will save you several weeks of difficult programming. So any effort involved in the planning process will usually be amply rewarded.

The real beauty of the planning stage, is that enables you to discover any serious flaws in your program early on. If some of these problems later prove to be insoluble, it's then usually possible to adapt your game so as to avoid them completely. You can also perform simple

experiments to check how the mechanics of your game will work out in practice. This lets you explore many possible game ideas before committing yourself to anything concrete.

2.2 Initial Ideas

Few computer games are written in isolation. Nowadays, most games can be placed into one of a small number of basic types. These categories include arcade games, adventures, or simulations. Some formats have been developed from just a single success story, while others are the latest developments of ideas which hark back to the very earliest days of computing.

Start off by choosing a game type which particularly interests you. Obviously, it's pointless to attempt to write an exact duplicate of someone else's program in AMOS Basic. Not only is this almost certainly illegal, but it's also completely futile. If people want to play the original Arkanoid, they will invariably buy the official version rather than your own. At least that way they will be rewarding the people who did the actual work on the project. Furthermore, if you wish to sell your program to either a commercial software house or a magazine, you will have to add something new and innovative to make your game stand out from the crowd.

A good source of inspiration is to look at as many existing examples of your chosen format as possible. You can then make out a complete list of the various things you like and dislike about these games. Hopefully, this process should give you a number of useful ideas about how the game might be expanded or improved. Think of different modifications to the game, and try to work out the consequences of these changes in terms of the final game play. Don't however, get too carried away with any single idea. Inevitably, as your plans progress, some notions will prove to be unworkable, and other possibilities will occur to you instead. That's half the fun!

2.3 Producing a Game Plan

I'll now go into the details of how one of these game plans can be created. The first stage is to produce a detailed specification. This should consist of a complete written description of the game in standard English. The aim is to list all the important features of your program, and to isolate any potential problems. You can also use this process as an opportunity to work out the detailed mechanics of the game play. What's the aim of the game? What are the rewards to the successful player?

Failure to concentrate sufficiently on this topic can have potentially disastrous results. It's possible to spend a great deal of time and energy to produce a game which looks and sounds brilliant, but which is completely unplayable. This is an area that even the experts can get wrong. In my experience the quality of the game play often bears no direct relationship to the quality of the screen effects. So it's vital to think carefully about the design of your game before you commit yourself to a lot of programming. Even if you already have a fairly good idea about the contents of your game, it's still worthwhile drawing out a detailed game plan. It's amazing how many improvements can arise from the simple transition of an idea onto paper.

2.3.1 Understanding the problem

I'll now show you how a complete specification can be produced for an actual game. I'll take the game Asteroids as an example. This is very similar to the Amosteroids program supplied on the AMOS Data disk. So it's well worth playing a game or two to familiarise yourself with the basic ideas before you continue. (It's called research!)

The first decision I'll take is to choose a name for my new game. The selection of a title is often quite difficult, since we'll need to find something which hasn't already been used. If you're stuck for original names, try using a Thesaurus. Although this sounds like an exotic

form of dinosaur, it's really just a dictionary of words organised by their meaning rather than alphabetically. Copies can be found in the reference section of your local library. Try looking up the word *Asteroid* for an example. You should be presented with a variety of alternative possibilities. For the time being, I'll choose the name *Pathfinder*, reserving the option to change it later if it proves necessary. I'll now write up a simple specification for this game.

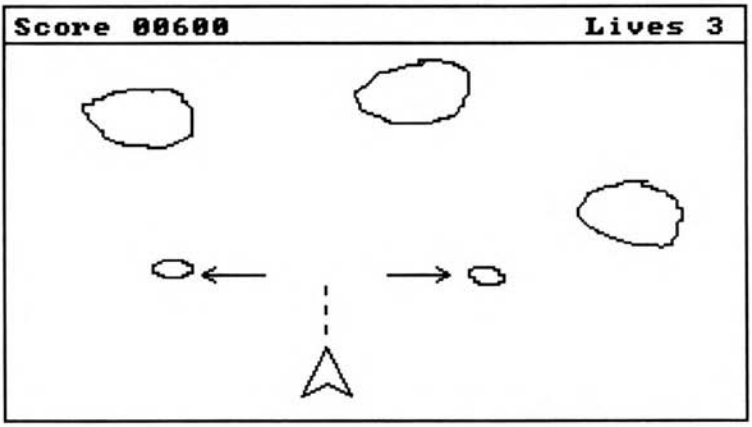

Figure 2.1: Mock-Up of a Pathfinder screen

Description:

This is a game based on the old arcade favourite ASTEROIDS. At the start, a small ship is positioned at the centre of the playfield. This ship can be moved on the screen using the joystick like so:

Left/Right: Turns the ship.

Up/Down: Increases/Decreases the thrust in the current direction.

Fire: Fires a salvo of missiles from the front of the ship.

As the game progresses, several large rocks slowly creep up from the edges of the playing area. The object is to destroy as many rocks as possible before they collide with the player's ship. This ship is provided with a missile launcher which fires a stream of deadly missiles at the approaching rocks. If the missile hits its target, the rock will be split into two smaller chunks. These will now move off rapidly in opposite directions. After a rock has been hit three successive times, it will be completely destroyed. A new rock will now appear from a random edge of the screen, and the game continues.

Alongside this verbal description, we can also add rough sketches of the various graphical elements which make up the game. A mock-up of a typical screen might look like Figure 2.1. We can now jot down a list of the various events which will occur during the course of a game, along with any initial thoughts about how these activities might be implemented in the final program.

Actions:

1. The rocks will move in straight lines from a random point on the edge of the screen. Notes: Implement each rock using a single Bob, and move via the AMAL Move instruction.

2. Shooting a large rock will split it into two smaller chunks. These will move apart at right angles to the original course. Notes: Large rock uses 32 by 32 Bob? Medium rock uses 16 by 16 image?
3. Destroying a medium sized rock will split it into two smaller ones. As before, the two rocks will move away in opposite directions. Note: Small rock uses 16 by 8 Bob?
4. Shooting a small rock will utterly destroy it, scoring maximum points.
5. The speed of a rock depends on its size. Smaller rocks move the fastest.
6. The points scored for destroying a rock varies on the size. The smaller the rock, the greater the points.
7. The player controls a small space ship with the joystick. Note: implement ship as 32 by 32 Bob?
8. The velocity of the player's ship can be changed by pulling the joystick up or down.

   Up increases the velocity, and Down reduces it.

   Note: Use something like Rockspeed=Rockspeed+Acceleration.

   Up represents a positive acceleration.

   Down reduces the speed by using a negative value for the acceleration.

   Just to make life easier for us, the movement direction will only be changed when the speed reaches zero. So although the players can turn the ship at any time, they'll need to stop it completely in order to change direction. This should simplify the mathematics considerably and add an attractive inertia effect to the game.
9. The ship can be turned by moving the joystick left or right.

   Note: Use a separate image for each direction.
10. If the ship is hit by a rock, both ship and the rocks are completely destroyed and the player loses a life.
11. Pressing the fire button releases a small missile which continues in the current direction.
12. The player is allowed to fire as many shots he or she likes.

If you examine these actions carefully, a couple of significant questions should arise. What will be the maximum number of rocks on the screen? This is important because the standard animation system only allows us to move 16 objects simultaneously. Although this limit can be exceeded using the SYNCHRO command (see Chapter 3), we will need to indulge a considerable amount of extra programming. It's essential to decide whether to use this system we are using well in advance, as it will save us a great deal of unnecessary programming.

It should be apparent that the total number of rocks on the screen will depend entirely on the original number of large rocks on screen. Each large rock will spawn exactly four smaller rocks during the course of the game. So the maximum number of rocks at any one time can be calculated from the following formula:

Maximum_rocks=Large_Rocks*4

In order to keep within the 16 object limit, we'd have to restrict ourselves to a maximum of 16/4 = 4 large rocks at the start of our game. This number does not, of course, include the Bobs needed for our other objects, such as the space ship. It therefore makes sensible to limit the initial number of asteroids to exactly three. This would use up 12 of the 16 possible animation channels, leaving us with three spares.

Alternatively, if we used the SYNCHRO system, we'd be able to manipulate a maximum of 64 objects simultaneously. Assuming we reserved four sprites for our space ship and missile, the number of large asteroids would be increased to a grand total of 15. Although this is technically feasible, it's debatable whether the game be playable with this number of asteroids. We'd obviously need to experiment carefully with the various starting conditions during the development of our game.

Another problem, is the nature of the missile. In the original specification, I've foolishly committed myself to an inexhaustible supply of ammunition. How is this going to be generated? Unless we're intending to use SYNCHRO, there would only be a couple animation channels spare. One solution might be to animate our missiles by hand. Remember that the only limit is in the number of animation channels. There are no such restrictions in the number of Bobs other than the amount of available memory. So if we run out of AMAL animation channels, there's nothing stopping us from moving our objects directly inside our Basic program.

Before preceding with our game plan any further, we could quickly check whether this approach was feasible by writing a small demonstration program. If we were successful, we could now design our game, secure in the knowledge that the solution to the problem was actually achievable. On the other hand, if we failed miserably, we'd need to devise some way of sidestepping this problem. The easiest solution would be to change the specification slightly. Supposing we removed the ability to fire multiple missiles (item 12). We would now be able to implement our missile using a single animation channel.

This situation is typical of the type of difficulty which is thrown up during the planning stage. If we hadn't bothered with the original game plan, we would only have discovered the problem well into writing the program. The only feasible solution might even have involved completely rewriting our entire game from scratch! Ouch!

Of course, the specification I have shown you is only intended as a simple demonstration, and would need a little more work before it was ready to proceed any further. The game play in particular, would currently appear to be dangerously repetitive. Here are a few suggestions for possible improvements:

13. Different types of rocks have separate colours.

14. Red boulders explode violently and score higher points.

15. Yellow boulders need several hits before they explode, and score maximum points.

16. If the player hits the spacebar, the ship jumps to a random location (Hyperspace option).

17. An enemy ship appears at random intervals and fires at the player. Destroying this ship scores extra points.

The key thing to remember about these game plans, is that they are only intended as a general guide. Even the best plan can be improved, and you will probably think up a number of interesting possibilities during the development process. So don't be afraid to experiment with your ideas as and when you think of them. Treat the game plan as a flexible framework which can be fleshed out in the course of time.

2.3.2 Modular programming

The next stage in the development process is to group the various activities into separate tasks. These will eventually form the basis for the procedures which will be used in the final AMOS Basic program. Experience has shown that any large program can be simplified enormously if it is split up into a number of smaller, more manageable chunks. This approach is known as modular programming.

Modular programming has proved phenomenally successful in the world of computing. That's because modular programs are both easy to write and easy to change. So we can start off with something extremely simple, and steadily expand it into a polished and attractive game. AMOS Basic includes a wide range of special instructions which support this system.

Subroutines

These are separate groups of instructions which start with a label and end with the RETURN statement. Subroutines can be placed anywhere inside your Basic programs. They are called from AMOS Basic using the GOSUB command. GOSUB is short for GOto SUBroutine. Here's an example:

```
NEW_SCORE:
Locate 30,0 : Print "Score:";SCORE;
Return
```

This routine would be executed from your program using a line like:

```
Gosub NEW_SCORE
```

Easy isn't it? Note: If a subroutine is accidentally run as part of your normal program, you'll be dropped back to AMOS Basic with an error message such as:

```
RETURN without GOSUB
```

You are therefore recommended to place all your routines in a separate section at the end of your main program. This can be proceeded with the STOP command to ensure that they are never executed by mistake. For example:

```
Main program
:     :
Stop
LABEL1:  Rem Subroutines go here
:     :
```

Procedures

Any AMOS program can be constructed out of a number of "Procedures". These are very similar to subroutines, but are COMPLETELY separate from your main program, and can include their own lists of variables and data statements. They are created with the help of the AMOS Basic PROCEDURE command.

```
Procedure NAME[list of variables to be loaded into the procedure]
:    : Definition goes here
:    :
End proc
```

Once you've defined one of these procedures, you can call it directly from your main program by simply typing its name. Here's a new version of the previous NEW_SCORE routine using procedures.

```
Procedure new_SCORE[SCORE]
     Locate 30,0: Print "Score:";SCORE;
End proc
```

This would be called from your program with a line like:

```
NEW_SCORE
```

Normally all variables used in a procedure are completely independent of your main program. So if you want to access the variables from your program, you'll need to either include them in the original variable list, or use the SHARE statement inside the procedure. For example:

```
Procedure NEW_SCORE
     Shared SCORE
     Locate 30,0 : Print "Score:";SCORE;
End proc
```

In practice, this restriction can make life rather difficult. If you need to access a lot of information from the main program, you'll probably find it easier to use a GOSUB instead.

Don't use the GLOBAL command, by the way. On paper, this is supposed to define a global variable in the main program which can be accessed immediately from inside any AMOS procedure. It's equivalent to adding a SHARE statement inside each procedure. Unfortunately, due to a slight bug in AMOS Basic, it's dangerously unreliable. In extreme circumstances, it can even crash your Amiga. Avoid!

Despite this, procedures are still pretty useful. Providing you use them for routines which are more or less self-contained, you're unlikely to have any significant problems. They can for instance, be neatly tucked away inside your main program with the Fold/Unfold command from the AMOS Editor (F9). This removes the procedure definition from your listing and replaces it with just the name. Let's fold our NEW_SCORE procedure and see what happens.

```
Procedure NEW_SCORE
     Shared SCORE
     Locate 30,0 : Print "Score:";SCORE;
End proc
```

This is replaced by simply

```
Procedure NEW_SCORE
```

You can of course, get the full procedure definition back at any time. Just move the cursor over the procedure statement and hit F9. If you're using a lot of procedures, you can exploit this feature to dramatically reduce the size of your program listings. This makes the structure of your program a great deal clearer, and allows you to concentrate on any problem areas, without having to scan through masses of irrelevant program lines.

The importance of being modular

The neatest thing about the modular approach, is that you can change your routines independently of the rest of your program. Suppose you were writing a screen scrolling arcade game. After you'd finished the program, you might have an terrific idea for an ultra-smooth scrolling routine. You could now replace the existing system with an improved version by simply changing your original definitions.

Another advantage of procedures and subroutines, is that they can be re-used again and again. So once you've created the perfect high-score table, you can incorporate it into dozens of games, with practically no additional effort. After a while most programmers quickly build up a whole library of useful routines. This allows them to concentrate on the interesting sections of their programs, without having to continually worry about boring high-score tables or save game features.

2.3.3 Splitting a program into its components

But how do we split up a large program into its individual components? The starting point is to make up a list of all the separate activities which are to be performed by the program. I'll now demonstrate this process using the Pathfinder game I mentioned earlier. The activities in this program can be grouped together in the following way.

- ❑ Initialise the screen.
- ❑ Move the rocks
- ❑ Move the ship
- ❑ Test the joystick
- ❑ Fire the bullet
- ❑ Detect collisions

Each of these activities represents the execution of a specific section of the program. Unfortunately, this list only gives us a very crude picture of the program, as there is no information specifying the particular order in which each activity will be performed. Before we can include this in your descriptions, we'll need some way of extending the existing notation so as to allow us to make decisions. The easiest way to achieve this, is to use something called pseudo-code (pronounced soo-doe-code).

2.3.4 Pseudo-code

Pseudo-code enables us to completely describe the action of any computer program using a limited form of English. Pseudo actually means *false*. So pseudo-code creates a false program which cannot be executed directly on your Amiga. The fundamental unit of this pseudo-code notation is the *statement*. Statements are represented by a few words which concisely explain a single operation in your program. The following English phrases are valid pseudo-code statements:

Move the ship

Update score

Like Basic, we can control our activities using statements such as IF...THEN, and loops like REPEAT...UNTIL. But since we are working on paper, there is no complicated syntax to remember.

Pseudo-code is not limited to just describing a computer program of course. Any process which can be split into a number of steps, can also be described using a list of pseudo-code statements. The best way to understand pseudo-code is to attempt to write some yourself. Try to produce a pseudo-code description of the activities involved in making a cup of black coffee. Assume you are giving instructions to an intelligent, but literal minded person who has no initiative whatsoever. If you have a little experience with adventure games, you should have a good idea of the sort of words you will need to use. After a little effort, you should have written something similar to the listing overleaf.

As you can see, even simple activities need to be broken down into quite a long list of instructions. Fortunately, writing a computer game is MUCH easier than making a cup of coffee! (How many coffee machines do you know which can perform all these actions by themselves?)

```
get coffee jar
get coffee mug
get spoon
grab top of coffee jar
 repeat
    Turn top
until top is removed
push spoon into coffee jar
scoop up some coffee
empty coffee from spoon into mug
put down coffee jar
put down spoon
put down mug
get kettle
fill kettle from sink
plug kettle's lead into power socket
turn on kettle
while kettle is turned on
If kettle boils
     Then
       turn off kettle
       pour water into coffee mug
     endif
wend
```

The aim of pseudo-code is to provide a sort of halfway house between a human language like English and a computer language like AMOS Basic. This allows us to produce a general description of how an activity can be performed, without having to worry about any boring implementation details. Eventually each pseudo-code statement will be translated into a series of AMOS Basic commands. After translation it may represent anything from one, to several hundred, actual instructions. Here are a few some small fragments of this pseudo-code applied to the pathfinder problem:

```
If rock is hit then destroy rock
If joystick has been moved
  then
    move the players ship
    check for collisions
  endif
Repeat
   Move rock
Until rock leaves screen
While missile is moving
   Check for collisions
Wend
```

The statement between the IF and THEN represents a simple comparison. If this comparison is fairly complex, each test may involve a separate procedure in the resulting program. This will be called before the check occurs, and will normally set a variable which can be tested using a single AMOS Basic IF instruction. Notice the indentation to highlight groups of associated statements. Also note the WEND and ENDIF instructions, which indicate the outer limits of the list of statements which are to be performed by the WHILE and the IF operations respectively.

It is important to realise that there is nothing particularly sacred about the above notation. Feel free to chop and change it as much as you like. If you would prefer to invent your own pseudo-code system instead of using mine, go ahead. The only thing that really matters is how easy it is for YOU to understand. Anything else is irrelevant. Now for a somewhat larger example of this pseudo-code, which concisely describes the action of my Pathfinder game.

Figure 2.2: Pseudo-Code description of Pathfinder

```
Initialise the screen.
Repeat
  Check for collisions
  If a rock hits the ship
   then
     destroy ship
   endif
  If a bullet hits a large rock
   then
      split rock
      update score
   endif
  If a bullet hits a medium rock
   then
      split rock
      update score
   endif
  If a bullet hits a small rock
   then
     remove rock from the screen
     update score
     add new rock from edge of the screen
   endif
  Test the joystick
  If joystick has been pulled UP or DOWN
   then
     change the velocity of the ship
   endif
  If joystick has been pulled LEFT or RIGHT
   then
     turn the ship
   endif
  If the fire button is pressed
   then
      activate bullet
   endif
    If bullet is activated
   then
      move bullet
   endif
    Move the rocks
    Move Ship
Until ship is destroyed
```

This pseudo-code is not supposed to be complete, and would need to be extended for the finished game plan. It is however, sufficient to enable you to isolate a few more of the game's components. Have a brief look at this description, and try to jot down a list of crucial activities. Hint: Each activity will be associated with just a single pseudo-code statement. The main activities will be:

- ❑ Initialise the screen
- ❑ Move rocks
- ❑ Move ship
- ❑ Detect collisions
- ❑ Destroy ship

- ❑ Split rock
- ❑ Update score
- ❑ Remove rock from screen
- ❑ Place new rock at edge of screen
- ❑ Test the joystick
- ❑ Change ship's velocity
- ❑ Turn ship
- ❑ Activate bullet
- ❑ Move bullet

Each of these activities can expanded into a separate list of pseudo-code statements. So the *Check for Collisions* activity could be expressed as:

```
Check for collision between a rock and the ship
 If bullet activated
     then
       check for collision between rock and bullet
       If rock is hit then
                If small rock then
                   destroy it
                Else
                   Split rock in two pieces
                   Set up movements
                Endif
       check for collision between ship and bullet
       If ship collides with bullet then destroy ship
     endif
```

We can now refine the game plan still further by expanding all of the procedures using the same systematic method. We may then isolate any new activities which will be required, and describe them with their own fragment of pseudo-code.

If we repeat this process, we will finally be left with a list of pseudo-code definitions for each of the procedures in your program. These can be translated directly into a small number of AMOS Basic instructions. Normally, this point will be reached within around three or four expansions. The game plan will now be complete, and we will have a full description of the workings of our game. Think of the planning process like the peeling of an onion, layer by layer. As the method progresses, the precise workings of the program will slowly be unravelled in front of our eyes.

My own example has practically reached this point. Only the *Destroy ship* activity will need to be developed any further. The various *Check for collision* routines could be performed using the appropriate BOB COL and COL instructions from AMOS Basic.

Sometimes our original plan will prove slightly impractical. If for instance, we had added a check for the collision between two rocks, we would have encountered a serious problem. This is because the AMOS Basic BOB COL command has to include the number of a specific Bob. We would therefore need to test the status of each individual object on the screen in turn, before a collision would be detected. This loop could slow down the action of our game significantly. We could avoid this complication by simply jumping back to the original specification, and making the appropriate changes. A complete explanation of this procedure can be found in Chapter 3 (Arcade games).

Remember that the specification can be easily altered at any point during the development process. It's not intended as a straight-jacket. Any elements which complicate your program without significantly contributing to the actual game-play should be mercilessly removed at this point. There's no point in making unnecessary problems, as these efforts won't be appreciated by the eventual player. Try to obey the golden rule of programming: *Keep it Simple!*

2.4 Critical Phases

The execution of any game can be split into three critical phases:

- Initialisation
- Control loop
- Termination.

The initialisation stage corresponds to the *Initialise screen* activity from Pathfinder. Its aim is to prepare all the various screens and load the variables with their initial values. The Control loop forms the heart the game, and controls the action of the entire program. A typical example of this type of procedure can be seen in the pseudo-code I showed you earlier for the Pathfinder program. (See Figure 2.2)

The Termination phase performs a simple cleaning up operation after the game has finished. This normally includes the following activities:

- Close any currently open files
- Update the high score table on the disk (Essential!)
- Present the player with the option of a further game.

2.5 Designing the Graphics

2.5.1 Choosing the Resolution

Unlike many other programming languages for the Amiga, AMOS Basic is capable of performing equally well in all of the Amiga's graphics modes. So it's essential to decide on the modes which will be used to construct the various areas of the display. These can be created with the SCREEN OPEN command. As you may recall, the format was simply:

```
SCREEN OPEN screen_number,width,height,number_of_colours,size
```

Where:

screen_number is an identification number from 0-7 of the new screen.

width and *height* set the actual dimensions of the screen in pixels.

no_of_colours specifies the maximum number of colours which can be displayed at any one time.

size selects the size of each individual screen point as either LOWRES or HIRES.

Each mode has its own unique set of advantages and disadvantages.

2 Colour Mode

This is mainly used for text displays such as high-score tables, and the text areas of an adventure. The advantage of using this mode is that an entire 320*200 screen takes up a mere 8K of memory! So there's lots of extra space for your text!

4/8 colour modes

These modes are perfect for the scrolling background in an arcade game like Defender because they take up very little memory. Blitter scrolling can be accomplished with consummate ease. Since the memory requirements are so small, you can make heavy use of the Double buffering system, without the risk of running out of memory on an unexpanded machine. These modes are often used in conjunction with the DUAL PLAYFIELD system. If eight colours seem a little restrictive, you can usually add extra colours using the AMOS RAINBOW command.

16 colour mode

A good all round display suited to most types of game. Sixteen colour mode provides acceptable screen effects at high speed. That's why it's used as a default in AMOS Basic.

32 colour mode

Although each screen takes a minimum of 40K, there are a lot of advantages to this mode. Not only are the screens much more colourful, but they can be redrawn reasonably quickly when required.

64 Colour mode (Extra-half-bright)

Whilst this mode is ideal for background screens, it's not really suitable for arcade games. Each screen takes a minimum of 48K, and this needs to be doubled if you're intending to use Bobs. Additionally, there's a noticeable slowdown in all graphics operations. These problems can be largely avoided using hardware sprites rather than Bobs. But be warned! Sprites are not for the faint hearted! Handle with care!

4096 colour HAM mode

HAM is only really useful for static displays such as those you'd find in an adventure. Because of the limitations of this mode, you can't use Blitter objects at all. This dramatically reduces the scope for animation effects in your games. On the plus side, HAM is capable of generating some tremendous pictures, and it's certainly worth consideration if you can live with the restrictions.

2.5.2 Working out the memory requirements of a screen

The memory requirements of an AMOS Basic screen can be calculated directly from the following equation:

MEMORY=(Width*Height*Planes)/8

Planes is the number of colour planes used in the required graphics mode. Table 2 contains the necessary values:

Table 2.1: Number of planes available in each graphic mode

Number of colours	**2**	**4**	**8**	**16**	**32**	**64**	**4096**
Colour Planes	1	2	3	4	5	6	6

Supposing you wanted to create a 320x256 screen using 32 colours. Looking at the table you will find that the number of screen planes is five. The amount of memory consumed by this screen will be:

320*256*5/8=51,200 bytes.

Note that if you're intending you use Blitter objects on your screen, you'll need to use double buffering. This system doubles the amount of memory required by your screens. So the final figure would be:

51,200*2=102,400 bytes! Wow!

2.5.3 Mock-ups

Before you begin drawing your final artwork, it is useful to sketch out the basic screen format on a piece of paper. Despite the undoubted power of drawing packages such as Deluxe Paint or Photon Paint, it's all too easy to get bogged down with extraneous details. You should initially concentrate on the overall structure of the screen, rather than producing a completely perfect rendition of the final image. Many games require you to split the screen into a number of separate sections. It is therefore important to decide on the physical size of these areas, and work out exactly where they will be positioned. Figure 2.3 is an example of a typical mock-up of a game screen.

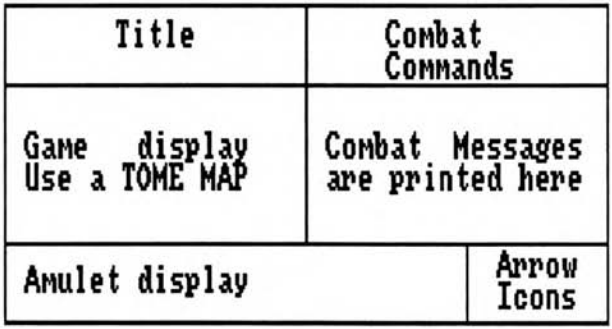

Figure 2.3: Mock-Up of a background screen

After you have created your mock-ups on paper, you can then enter your sketches directly into your favourite drawing package. These can be saved on the disk, and used for the preliminary versions of your game.

It may be tempting to spend ages drawing the finished artwork at this point. This is usually a very dangerous idea, as it fixes the structure of the graphics at a far too early stage. If you keep the initial screens as simple as possible, you can effortlessly make major modifications to your program during the development of your game.

2.5.4 Designing the screen objects

Any sprites or bobs which are to be used need to be chosen with special care. Although AMOS Basic is extremely good at moving your objects around on the screen, some limitations are inevitable. Here is a list of the various constraints:

- The smaller the object, the faster it can be moved across the screen.
- The maximum speed of a Bob depends on the screen resolution. There's a noticeable speed difference between an eight colour bob and a 32 colour bob of the same size.
- The more objects you attempt to animate, the slower they will move.
- Hardware sprites can be moved faster than Blitter objects, and work equally well in all resolutions, including HAM.

- Hardware sprites are restricted to either four or sixteen colours. Hint: The existing Sprite Editor is not capable of drawing 4-colour images in low resolution. So use High resolution instead. The images will be exactly the same.
- Bobs can be positioned anywhere on the screen. Sprites can only be displayed in a limited number of combinations. See the AMOS Basic manual for a detailed explanation of this system. It's pretty complicated.

2.5.5 Memory requirements of Bobs and sprites

It's sensible make a rough calculation of your memory requirements during the initial planning stage. This allows you tailor the size of your various screen objects precisely to the amount of available memory.

Memory requirements of a Blitter Object

The biggest drawback with Blitter objects is that they are incredibly memory hungry. Bobs force you to use the double buffering system, which effectively doubles the amount of memory required to store your background screens. So if you're using a 320x255 32-colour screen, you'll need at least 100K of CHIP memory just for the display. In addition to this figure, each Bob on the screen requires its own separate chunk of memory. This can be found using the equation:

MEMORY=(Width*Height*Planes)*3/8

Planes is the number of screen planes in the object. See Table 2.1.

If you were using a bob of dimensions 64x64 in 32 colours, you'd therefore need:

(64*64*5)*3/8 bytes

or 7680 bytes.

When you add in the memory used by the screen, you get a grand total of 110K. This would leave you with very little memory for your program on an unexpanded A500.

Note that the only limit to the size of your Blitter object is the amount of available chip memory. If you've enough memory, you can create bobs as large as you like. I've personally created objects up to 320x200 in 32 colours without any problems. Amazingly enough, these massive objects could be moved around the screen surprisingly quickly!

Memory requirements for hardware sprites

Compared to Blitter objects, hardware sprites are remarkably efficient. The memory consumption can be calculated from the following simple formula:

Memory=(Total sprite length)*96

The total sprite length ranges from 16 to 255, and can be set using the SET SPRITE BUFFER command. Even at the maximum size, the sprites only take up around 24K. Furthermore, there's no need to use the double buffering system, so there's a dramatic saving in your program's memory consumption. Against this, you need to take into account the restrictions regarding sprite positions.

Although you can display up to 47 sprites on the screen, the total width of all the sprites cannot exceed 64 pixels on any horizontal line (128 for 4-colour sprites.)

2.5.6 Making a mock-up of a sprite

As with the background screens, it's sensible to produce a complete mock-up of all your objects before you create them with the sprite definer. This is especially important if you wish to produce an animation sequence such as a walking man. By keeping the initial drawing extremely simple, you can work out the position of the various limbs relatively easily. You can then draw successive frames of the movement on separate pieces of paper, and animate the figure by simply flicking through the pages one after another.

This system may seem rather crude, but despite the availability of modern computers, it is still commonly used by many professional animators. After you've animated the skeleton of the figure, you can then enter these frames directly into the AMOS Basic sprite definer. It's wisest to leave these in their original state until the program is nearing completion. This will allow you to quickly change the size or appearance of your sprites, as your game evolves.

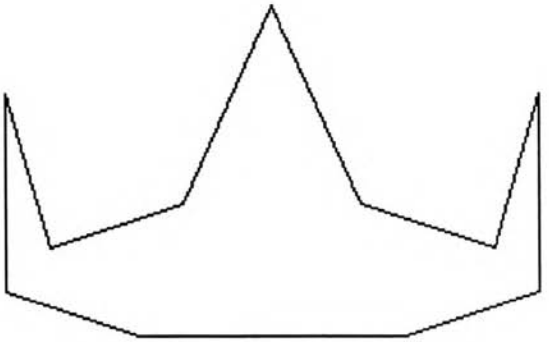

Figure 2.4: Mock-up of a sprite

2.6 Data Structures

2.6.1 Introduction to data structures

The last phase of the planning process is to decide where the various pieces of information required in your game are to be stored. If you have written out your program in pseudo-code, some of this data will be fairly obvious. You can for instance, quickly list many of the simple variables which will occur in the Pathfinder program. For example:

```
score number_of_rocks lives_left
```

There will however, also be a need for far more complicated arrangements of data. These will be held in special data structures. A data structure is just a fancy name for a collection of similar information. There are a number of different types of data structures which can be used, including arrays, lists, stacks and records. When writing a computer program you are often faced with a bewildering range of storage methods. By choosing the most appropriate data structures to the problem in hand, you can often speed up your game significantly.

Supposing you wished to select the data structures required for the Pathfinder program. In this case, the biggest requirement will be for a way of keeping track of the individual rocks. Every rock will be drawn using a single Blitter object.

AMOS Basic refers to each Bob using a number from 1 to 63. It's very important to keep track of the size of a rock, as this will be needed to decide whether the rock should be split in two or destroyed after a successful hit. You therefore need to create a table containing the size of each individual rock. This can be done using a simple array which can be defined at the start of the program like so.

```
DIM ROCK(12)
```

Each bob will have a number specifying the size of the rock which it represents. If you have decided to have different sorts of rock in the game, you will also need separate arrays to hold this data. These can be defined in exactly the same way.

When you are defining your variables, you may be tempted combine many similar arrays in a single multi-dimensional array. It's recommended that you use a single dimension whenever possible, as AMOS Basic does take slightly longer to retrieve information from an array with more than one dimension.

This means that a line like: `X=XARRAY(10):Y=YARRAY(10)` performs marginally faster than the equivalent: `X=ARRAY(10,1):Y=ARRAY(10,2)`

The first line is also much easier to read.

2.6.2 Memory constraints

When you are choosing the data structures used by your game, it's important to keep an eye on the amount of memory they will consume. The storage used by any variable depends on its type.

Type	Size	Notes
Integer	4 Bytes	Default variable type
Real	4 Bytes	Followed by a #
String	Up to 65536	Followed by a $

2.6.3 Memory requirements of an array

The memory taken by an array can be calculated by simply multiplying the number of elements by the size. In practice, the memory used can quickly mount up, especially when you are using arrays with more than a single dimension. Take the array:

```
Objects(10,10,10)
```

This consumes a total of 10*10*10*4=4000 bytes. If you had dozens of these arrays, you could easily run of memory, especially on an unexpanded A500. Warning! As a default AMOS only allocates around 8000 bytes for your variables. If your variables exceed this requirement, you'll get an *Out of memory* error. This does NOT necessarily mean you have run out of memory. It only tells you that the current variable area is too small. You can expand it by placing a SET BUFFER command at the start of your program. For example:

```
SET BUFFER 30
```

reserves 30K for your variables.

2.7 Overview of the Game Plan

The planning stage of your game is now complete. Whew! If it all looks rather tedious, it is important to realise that the entire process can often be finished in less than a day. Pseudo-code

in particular, is much easier to write than Basic, as you can leave out many of the fiddly programming problems and concentrate on the overall structure of game. Even if you are only prepared to spend a couple of hours on your plan, you can still save yourself weeks of futile programming.

I'll conclude this section with a summary of the various steps in the planning process.

1. Choose a game type
2. Produce a written specification. This should include mock-ups of any screens used, and brief descriptions of the size and appearance of the sprites.
3. Isolate the various activities in your program.
4. Generate a pseudo-code description of each activity starting with the control loop.
5. Successively repeat steps 3-4 for each of the activities in your program.
6. Isolate the information needed by your program, and choose the appropriate data structures to contain it.

2.8 Using the Game Plan

Once you've finished your game plan, you are now ready to start generating the program lines which make up your game. Normally, this should be fairly easy, because much of the work will have already been done. It's best to write each section of your program on paper before typing it in. This is especially useful if you don't have a printer, as it can often prove almost impossible to debug large programs without a listing beside you. In case you're wondering, Debug is piece of computer jargon. The errors in a program are known as Bugs. So debugging refers to the process of correcting and testing a program to remove all the errors.

As you write your program, it's sensible to make a note of all the different variables on a separate piece of paper. This will serve as a valuable memory jogger during the debugging process. Don't bother with creating the finished artwork until the program is approaching completion. Try to concentrate on the game play without the distractions of the graphics.

If you're producing an arcade game (Chapter 3), most of the programming effort will need to be spent on producing the sections which will perform the object movements. Even if you have planned your game to perfection, there will inevitably be some problems which you simply cannot solve on paper. You then need to experiment with your program until you have achieved the desired effect.

Take the Pathfinder program. In this case, you will have to concentrate on the sections of the program which move the rocks around on the screen. This will require you to create a list of AMAL commands which will be used to animate your objects on the screen. Although you may have a very good idea about the approximate directions you will need, you will still need to experiment with a number of possibilities before you can generate the precise results you desire.

It's therefore a good idea to start off with a simple test program which will enable you to play around with the various different options. Overleaf, there is a small example of a sprite experimenter program for you to type in.

This program expects to find some sprites in memory before it is run. If you've already created your sprites, you can enter these using a line like:

```
load "sprites.abk"
```

Otherwise you should use the sprite definer accessory to produce a set of sprites with the appropriate sizes. These don't need to be anything fancy, as you are only interested in the speed and direction of the sprites rather than their appearance. Often a simple rectangular block is more than sufficient for this purpose. After you have experimented with this program, you should be left with a list of movement definitions which can be incorporated into your game. Personally, I use this prototyping technique a great deal. By testing a bare-bones version of the program, you can quickly determine which approaches will actually work, and which can be safely discarded.

Example 2.1: Sprite Experimenter program

```
Do
cls
input "Sprite Number";S
input "Image Number ";I
line input "Movement commands";M$
sprite S,160,100,I
Amal S,M$:Amal On
input "Stop?";S$ : Amal off Loop
```

2.8.1 Collision detection

The most critical part of an arcade game is the collision detector. If this is badly written, your game will be virtually unplayable, no matter how good it looks. You should therefore endeavour to make this routine as fast and reliable as possible. Remember that these instructions all work relative to a sprites hot spot. So choose the position of this point very carefully. For further information on the subject of collision detection, see Chapter 3.

2.9 Drawing the Background Screens

It's now time to design the various screens which will be appearing in your game. AMOS Basic provides you with a range of facilities which make it especially easy to incorporate beautiful graphics directly into your Basic programs.

Although AMOS doesn't supply you with a built-in drawing utility, it's totally compatible with the standard IFF image format. This is currently supported by the vast majority of drawing packages available on the Amiga, including famous programs such as Deluxe Paint and Photon Paint.

The single most important consideration when designing your graphics is to make them as easy to change as possible. The majority of software houses now employ separate people to write their games, and to draw the various screens. This has led to a dramatic improvement in the quality of these graphics over the past few years. If you know someone who can draw, it's well worth persuading them to help produce your various screens and game objects. It's important to stress that computer experience is NOT essential for this purpose. All that's needed is the basic artistic ability. Most drawing packages on the Amiga are extremely easy to use, and many artists will get to grips with them surprisingly quickly. Currently there's a big demand for experienced computer artists. So if your game is successful, it could be the stepping stone for a whole new career for your friend.

A few words of warning though. If your friend isn't a programmer, you'll need to lay down a few simple ground rules. This will ensure that you can use the resulting images in an actual game.

- If you are creating a series of tiles for use with a map definer such as TAME or TOME, all the blocks should be exactly the same size. Otherwise they will be completely useless.

- The widths of all sprite or Bob images should always be a multiple of 16 units wide. It's important to keep this in mind if you are drawing your image with an external drawing program such as Deluxe paint III.
- The fewer colours used in your screens, the faster the resulting movement effects. So don't create 64 colour images for use with an Arcade game. Remember, sprite images can include a maximum of 16 colours, and Bob images are limited to anything up to 32 colours.
- Any animation sequences should be kept as short as possible for maximum speed.

If you wish to sell your game commercially, it is quite likely that your artwork will need to be re-drawn professionally at some stage. Even if your graphics are exceptional, there is always the possibility that the name will be have to be altered before publication. Most commercial games go through several different name changes during development. This could have a significant affect on the final appearance of your game.

Fortunately, AMOS Basic is amazingly flexible in this respect. All the sprite definitions are stored in a separate memory bank, and can be changed independently of the rest of your program. Screens can be loaded from the disk at any time using the standard IFF format. They can be compressed into just a fraction of their normal size via a range of built-in compaction commands. This system is perfect for your title screens and for the general background areas for your game, as it produces images of the highest possible quality.

The game screens are a different proposition altogether. Most modern arcade games include vast numbers of the different levels. This is especially true of screen scrollers like Xenon II or Bloodmoney. If you want to incorporate these effects into an AMOS Basic game, you'll have to compact each screen into mere hundreds of bytes. This is clearly impossible using the standard AMOS compaction routines.

The solution is to make use of a map definer utility. The AMOS Program disk contains a program called TAME program for this purpose. It constructs your images out of a series of small rectangular building blocks. These blocks are created by cutting a standard IFF format screen into dozens of smaller units. Each block can be reused again and again, to generate hundreds of levels. With the help of this system, you can often cram entire screens into just 60 or 70 bytes. So there's no real limit to the potential size of your playing area.

Although TAME is undoubtedly useful, it's certainly not perfect. It's actually pretty tame! That's is not just my view. It's shared by the man who wrote it, Aaron Fothergill. Sadly, Aaron was forced to rush the original TAME out to meet an impossible publishing deadline. He's therefore pulled out all the stops to produce a new map editor known as AMOS TOME, which is simply oozing with power.

Unlike TAME, AMOS TOME is incredibly easy to use, and comes with its own built-in drawing system. It's so good, in fact, that I've devoted an entire section to it in Chapter 7. This provides a detailed explanation of how it can be used to create the fast scrolling backgrounds required by an Arcade game or RPG.

2.10 Documentation

If you've ever attempted to modify one of your programs after a period of some months, you will already be aware of the vital importance of good documentation. This is especially true if you are hoping to sell your game. Frankly, a badly documented program is unlikely to be accepted by any of the magazines, as the workings will appear to be incomprehensible to their readers. Usually software houses will often require substantial alterations to a game before accepting it. So if your program is resistant to change, it could easily be rejected out of hand.

Some people restrict documentation to the occasional REM. There is however, a lot more to program documentation than this. Each subroutine in your program should begin with a couple

of REMs explaining precisely what it does, and detailing any variables it uses to communicate with the rest of the program. If you have made a game plan, this will form a valuable part of your documentation. So tidy it up and update it to provide a full description of the program structure of your game.

If you try to add the remarks after the program has been written, you can easily end up missing something crucial. Whenever possible, you should try to choose the names of your variables to indicate the nature of their contents. This massively improves the readability of your listings. Unfortunately it also makes them harder to type in! Look at the following lines of Basic code.

```
If X5=5 Then CSC=CSC+100
If SH=5 Then SC=SC+100
If SHIP=5 Then SCORE=SCORE+100
```

All lines perform exactly the same, but the second and third versions are a great deal easier to read.

2.11 Testing

2.11.1 Bug hunting

After you've written your game out on paper, it's time to enter it into your Amiga. Ideally your program will work perfectly the first time you run it, but unfortunately this is very rarely the case. You can divide the various bugs in your game into three separate categories.

Syntax errors

The easiest problems to solve are caused by syntax errors. These generate error messages during the testing process, and are usually produced by simple typing mistakes which can be quickly corrected.

Run time errors

Some errors can only be detected when a program is actually running. These occur when you attempt to enter stupid values into an AMOS Basic instruction. Take the following program:

```
X=-1 : Y=-1 Locate X,Y : ? "This line doesn't work"
```

When you run this program, you get an *Illegal window parameter* error on the second line. That's because the LOCATE instruction only works with positive values. Of course, the above example was rather trivial. In reality the contents of your variables would be set to the wrong values due to an error in one of your calculations. This calculation could be hundreds of lines away from the actual error line.

Logic errors

The last type of bug is the dreaded logic error. These are created by an error in the design of the program. If you have planned your program properly, they should be rare. But since no-one is infallible, it is quite likely that the odd logic error will still creep into your program unawares.

The only solution is to completely re-write the section of your program which has failed. If you've split your program into a lot of small sections, you'll probably be able to get away with replacing a single routine. If however, you've created your program in one unwieldy chunk, you may be forced to make major changes to the entire program. This could take you weeks of

arduous programming. That's another reason why good programmers make heavy use of procedures and subroutines.

2.11.2 Debugging techniques

Whenever possible, you should always try to test each section of your program separately. This removes the additional complication of debugging the entire program, and isolates any problems to a single section of code. Obviously, this approach is only effective for some types of routines. Typical examples include subroutines for input, object movement, screen scrolling, screen generation, adventure parsing and character generation. These can be entered into the Amiga one at a time, and saved to the disk in individual files. Other activities however, can only be tested within the context of the finished game. The classic example of this sort of procedure is the collision detection routine used in an arcade game.

Debugging is an art, and it's well worth explaining in some detail. The testing process is split up into three separate phases. These can be summarised by the questions: WHAT, WHERE, and WHY?

WHAT is the problem?

Before you can correct an error, you first need to be able to isolate it. Unless you can reproduce the error at will, you'll have no way of testing your various solutions. So it's vital to work out which activities generate the error and determine as precisely as possible what is going wrong. Here's an example for you to type in:

Example 2.2

```
rem X marks the spot!
rem Place a joystick in the LEFT socket
rem Hold down the fire button to print a character.
Do
     If Jup(1) Then Dec Y
     If Jdown(1) Then Inc Y
     If Jleft(1) Then Dec X
     If Jright(1) Then Inc X
     Wait 1:Rem slow down the loop so you can see the results!
     rem only move the cursor if the joystick has been pulled
     if OX<>X or OY<>Y Then Locate X,Y : OX=X: OY=Y
     rem If button has been pressed print X at cursor position
     If Fire(1) Then Locate X,Y : ? "X";
Loop
```

The preceding program contains a small, but annoying run time error. Play around with it for a while and see if you can spot the problem. Did you spot it? When the cursor is moved off the edge of the screen, you get an *Illegal window parameter* error. We are now ready for the second stage of the debugging process.

WHERE is the bug?

The next step is to find the approximate area of your program which is generating the error. Ideally, you want to be able to isolate the problem to just a couple of lines of instructions. In the case of run-time errors, this is usually trivial, as AMOS will display the offending statement on the screen. But logic errors are much more difficult to track down. The best approach is to place several *breakpoints* at the important points in your program. These display the current position on the screen, and wait for you to hit a key before proceeding. You can now page through your program a bit at a time, making a note at the point your error occurs.

AMOS Basic provides you with a useful FOLLOW command for this very purpose. It uses its own separate screen, so it's perfect for debugging Arcade games. The format is:

FOLLOW [list of variables separated by commas]

When the FOLLOW command is executed, a small screen will be created and the current instruction will be automatically displayed, along with contents of the selected variables. This screen can be moved around using the UP/DOWN arrow keys. You can then hit a key to step through your program one instruction at a time. Here's an example:

```
Follow I
Do
For I=32 To 255 : ? chr$(i) : Next I
Loop
```

As you can see, FOLLOW generates a great deal of information. If this is unwanted, you can create a single breakpoint using the commands:

```
Follow : Follow off
```

This halts the program and just displays the present line. The program continues normally when you hit a key. Warning! FOLLOW is not compatible with the CLOSE EDITOR command! FOLLOW is an incredibly useful instruction. Pity it didn't make the original AMOS manual though!

Let's get back to our previous example (2.2). When we run this program and trigger the error, AMOS neatly informs us that the problem occurs at the Locate X,Y instruction on the line:

```
If OX<>X or OY<>Y Then Locate X,Y : OX=X : OY=Y
```

WHY does the error occur?

This is the final step in the debugging procedure. Once we've worked out precisely why the problem is occurring, it can usually be fixed in matter of seconds. Sometimes the cause of problem will be due to a simple typing mistake. Supposing you intended to type:

```
Print "Hi there!"
```

If you actually entered:

```
Prant "Hi there!"
```

you'd naturally get an error. Usually, however, the problem lies with the contents of your variables. These can be checked with the help of the FOLLOW command.

Take the program in example 2.2. This could be tested with:

```
Do
     If Jup(1) Then Dec Y
     If Jdown(1) Then Inc Y
     If Jleft(1) Then Dec X
     If Jright(1) Then Inc X
     Wait 1: Rem slow down the loop so you can see the results!
     rem Note the FOLLOW command
     if OX<>X or OY<>Y Then Follow X,Y : Follow off:Locate X,Y :
     OX=X : OY=Y
     rem If button has been pressed print X at cursor position
     If Fire(1) Then Locate X,Y : ? "X";
Loop
```

If you run the new program, you should notice that the problem only occurs when X<0 or X>39 or Y<0 or Y>24. You should now have a good idea as to the cause of the error. LOCATE only works if the coordinates lie within the current screen window. Example 2.2 ignored for this possibility completely. So it's not surprising that it failed when you moved the cursor past the screen boundaries. There are two possible solutions:

❑ Add a check for X and Y using an explicit IF statement:

```
If X>0 and X<40 and Y>0 and Y<40
     If OX<>X or OY<>Y : Locate X,Y : OX=X: OY=Y : endif
     If Fire(1) : Locate X,Y:? "X"; : endif
endif
```

❑ Limit the values of X and Y using:

```
If Jup(1) and Y>0 Then Dec Y
If Jdown(1) and Y<24 Then Inc Y
If Jleft(1) and X>0 Then Dec X
If Jright(1) and X<39 Then Inc X
```

Admittedly, the above problem was pretty obvious. If the bug is more intractable, you'll need to rely on two additional techniques:

Backtracking

Most run-time errors are caused by placing the wrong values into one or more variables. Once you've discovered the offending variables, you can use the AMOS FIND command from the editor to track down all occurrences in your program. Don't forget to open any previously closed procedures before you start. You can now insert explicit FOLLOW commands at the points in your program where the variables are set. This should allow you to find the precise area where the problem has occurred.

Breakpoints and single stepping

Logic errors often require a great deal of thought before they can be solved. Insert breakpoints around the critical point in your program and note down the contents of important variables. If you've got a printer attached, you can print out all the intermediate values on paper using LPRINT.

The worst problems are caused by synchronisation errors. So make sure you've inserted WAIT VBL instructions after commands like SCREEN SWAP. If you're really unlucky, you may need to step through your routine one instruction at a time.

Always make a backup of your program before you make any serious changes. This will allow you to get back to the original version of your program if your new approach doesn't work.

If all else fails, you'll probably have to resort to pen and paper. Take the place of the computer, and execute each line of the program by hand, jotting down the results as you go. This may seem unbearably tedious, even with the aid of calculator, but it's capable of solving problems which would be impossible in any other way. That's because it concentrates your mind entirely on the current difficulty, without the distractions of the TV screen or computer. If the program involves animation, try sketching each frame on a piece of paper. You may find you've missed out something really crucial. Movement can be tested with the aid of simple models on a desk, or appropriately shaped bits of paper.

Finally, I'll say a few words about errors in your AMAL programs. If your AMAL routine is relatively short, you'll probably be able to fix it by hand.

Here's a useful debugging routine for you to type in:

```
Procedure DEBUG_AMAL
     Synchro Off
     While Key State(95)=0: rem HELP key aborts
          Wait Key
          rem step through your AMAL program
          Synchro : Wait Vbl
     Wend
     Synchro on
End proc
```

Place a call to DEBUG_AMAL after you've initialised your animation string. You'll now be able to execute your AMAL program a step at a time.

Larger programs may need the help of the separate AMAL editor supplied on the AMOS Extras disk. This can be rather fiddly to use, but it does contain a powerful monitor program which allows you to single step through your AMAL routines an instruction at a time and examine the various AMAL registers. So if you discover a problem in your AMAL program, you'll usually be able to track it down reasonably quickly.

2.11.3 Final testing

This is the last stage in the creation of your game. It is also one of the most critical. It's vital to check every possibility of your game for errors, especially if you are considering selling it.

The final testing should be exhaustive. Failure to do this can have potentially disastrous effects, as I know from painful experience. So try to do something really silly in your game, and see how it reacts. Believe me, if there's any conceivable way a user can mess up your program somebody will probably attempt it! The only safeguard against this, is to be incredibly thorough in your testing. It's also a good idea to make use of the AMOS Basic ON ERROR directive to allow your program to safely recover from any unavoidable errors, such as the player unexpectedly removing a disk while your program accessing the drive.

Don't assume that the player has any previous knowledge of the game at all. Remember that many people attempt to play a game straight from the box, without bothering to read the instructions. What's more, such people include many professional software reviewers, who often have to work to very strict dead-lines.

If you're writing an arcade game, concentrate on the critical routines such as collision detection. These are almost impossible to test in isolation, and need to be checked very carefully indeed. The biggest problems with these routines is caused by the fact the object movements are performed independently of the rest of the program.

Unless you execute the collision detection command at exactly the same time as the collision occurs, the result could be out-of-date. This means that your program needs to call the collision detection routine at frequent intervals during your game. Fortunately, AMOS includes a range of the collision detection commands which allow you to perform most tests almost instantaneously.

The last aspect of your game which you should check is the quality of the game play. This is really a very subjective area indeed, and can only be checked by getting as many people to play the game as possible. Feedback is essential, as many problems will only come to light when somebody else is running your program.

In my experience, there is a very fine line between a game which is exciting and challenging, and one which is impossibly difficult. Sometimes by reducing the speed and complexity of a game you can vastly improve its appeal. So don't be afraid to modify the game extensively to get the feel just right.

2.12 Optimisation

I'll conclude this chapter with a brief look at a number of optimisation techniques which can be used to improve the overall performance of your AMOS Basic programs. Some types of game require your program to execute a large number of complicated calculations time and again. These calculations can take exorbitantly long to perform, and this can slow down the action of your game considerably.

2.12.1 Look-up tables

The classic solution to these problems, is to work out the most commonly needed results in advance, and store them in a so called look-up table. This is just a list of pre-computed values which can be subsequently accessed virtually instantaneously from your game. The best way to create one of these tables is to write a small Basic program. Here is a crude demonstration of how one of these tables could be produced.

Example 2.3: Table of sines

```
rem Table of SINE values
Degree
Dim _SIN#(360)
for S=0 to 360
_SIN#(S)=SIN(S)
Next S
```

You could now generate the SIN of 45 degrees with a line like:

```
x=_SIN#(45)
```

Look-up tables are often converted into data statements and inserted directly into a program.

2.12.2 The shift instructions

Another useful optimisation technique, is to replace certain arithmetic operations with faster versions. You may already know that INC and DEC can be used to add or subtract one from a variable at high speed. Swapping X=X+1 with inc X, and X=X-1 with DEC X can improve the performance of critical sections of your program by a factor of two! Similarly, using ADD V,N instead of V=V+N will speed up many calculations by around 50 percent.

There are also versions of "*" and "/" which provide faster multiplication and division operations. These are available using the ROL and ROR instructions.

ROL multiplies a variable by a power of 2.

In other words:

ROL.L 1,X is equivalent to X=X*2

and:

ROL.L 2,X is the same as X=X*4

Similarly, ROR will divide a number by a power of 2.

ROR.L 1,X is identical to X=X/2

ROR.L 3,X can replace X=X/8

Typical speed improvements are up to 25% for ROL and over 80% for ROR. One minor limitation is that both of these operations will only work for positive numbers smaller than 1,073,741,824. This is unlikely to be a serious restriction!

2.12.3 Optimisation check-list

- ❑ If your programs perform many complex computations, work them out in advance and place them in a look-up table.
- ❑ Replace statements such as A=A+1 and A=A-1 by INC A and DEC A respectively.
- ❑ Replace statements like A=A+10 with the equivalent ADD command.
- ❑ Replace divisions and multiplications by a power of 2 by ROR.L and ROL.L instructions.
- ❑ When possible, use explicit constants in your assignments instead of variables. A=A+10 is marginally faster than A=A+TEN.
- ❑ FOR..NEXT loops are usually faster than WHILE..WEND or REPEAT..UNTIL loops.
- ❑ Don't attempt to cram several statements on a single line. Unlike most versions of Basic, AMOS performs better when there's just a single statement on each line. So a line like;

```
Inc X: X=X+Y : Gosub 1000
```

would be slower than:

```
Inc X
X=X+Y
Gosub 1000
```

If speed is of the essence, replace critical procedures with the equivalent GOSUB ... RETURN statements. GOSUBs aren't quite as powerful as procedures, but they are considerably faster. The conversion process can usually be accomplished in a matter of seconds.

If you still have speed problems, you will probably need to purchase the AMOS Basic compiler from Mandarin/Europress. This is capable of making many AMOS Basic programs execute up to twice as fast, and is well worth a look. Oh, and the manual's pretty nifty as well. (Guess who wrote it!)

2.13 Conclusion

In the next few chapters, I'll be providing you with a host of cook-book solutions which will hopefully cover most of your common programming needs. So read on, and discover the exciting world of games creation with AMOS Basic.

3

Arcade Games

3.1 Space Invaders

All modern arcade games are descended in some small way from a single classic program. The original Space Invaders game confronted the player with a menacing wave of alien space ships which were marching ominously towards the bottom of the screen. As the game progressed, each alien would let off a salvo of deadly nuclear missiles which would gradually eat away at the player's defences. The aim was simply to destroy the invaders before they reached the bottom of the screen. Easy eh?

From our current perspective, it's impossible to understand the phenomenal effect of this game on the early 70s software scene. Nowadays, Space Invaders would be treated with some derision, as it compares badly with even the worst arcade games available for the Amiga. But remember, those were the dark days of computing, when the Commodore Amiga was nothing more than an impossible dream. Space Invaders was the first computer game most people had ever seen, and it managed to capture the public's imagination in a way which has never been successfully duplicated.

For me, Space Invaders triumphed because of the compelling nature of the action. Anyone who wanted to achieve any respectable score would be unable to relax for a second along the way. Unfortunately, after you had played the game for some time, the movements of the alien ships started to become depressingly predictable. So although the game never became easy, it slowly but surely, lost its appeal.

Later developments such as Galaxians, were to incorporate a much needed element of variety. In order to complete one of these games, you had to defeat a vast number of different screens, each with its own unique set of challenges. Galaxians also moved the aliens in smoother and more realistic attack formations. This tremendously improved the overall quality of the game, because you never knew precisely where the aliens were going to appear next.

Another of my favourites was Gorf, which included some surprisingly modern features. It was for instance, the first arcade game to provide the player with a computer generated speech system. Ok, so it wasn't exactly intelligible. It actually sounded like a spaced-out Dalek with a serious speech impediment! Think of the Amiga's speech system and reflect. Believe it or not, this was WORSE! Oddly enough, it was considered a real innovation in its day! Gorf was effectively a clone of both Space Invaders and Galaxians, and it attempted to combine four entirely separate arcade games in one. This idea caught on quickly, and has now become a standard feature in many popular games.

Up to now, the only way of creating one of these games has been to write the entire program in complicated 68000 machine code. Thankfully with the release of AMOS Basic, these days are finally over. Francois Lionette has cleverly taken care of all the messy hardware details and provided us with everything we need to produce playable arcade games in the comfort of our very own homes. It's rather like having a brilliant machine code programmer permanently on tap!

3.2 Designing a Shoot-'em-up

As always, it's essential to plan the design of our game before we consider programming. In the final analysis, the success or failure of a game depends on the ideas behind it. So, unless we expend a little time and energy into the initial design, we won't have a hope of creating anything worthwhile. In fact, we might as well give up and spend our time zapping aliens in Xenon II! It'll be much more rewarding!

I find it useful to begin with a written specification. This crystalises the aims and aspirations of the program, and gives the perfect opportunity to play around with various possible ideas before we commit ourselves. The specification isn't essential, and it certainly doesn't need to be written in particularly good English. Rough notes are fine, as we can always tidy them up later for our final documentation.

The only crucial requirement is that we THINK about the game in detail, and work out in advance precisely what we are trying to achieve. If we plan our programs carefully, we'll be in an excellent position to write an enjoyable and challenging game. Otherwise, we'll be simply wasting our time! Just to start the ball rolling, I'll begin with a brief discussion of the various possible types of Arcade games.

3.2.1 Types of arcade games

Arcade games can be divided into four main categories:

Static shoot-'em-ups

Typical example: Galaxians/Gorf

In a static game, all the action takes place on a static background screen which remains fixed for the duration of the program. Variety is added by introducing different types of invaders with ever more complicated attack formations.

Although static games may sound a little old-fashioned, they are extremely easy to produce in AMOS Basic. What's more, the game-play is practically identical to the more advanced types. Providing we use a little imagination when we are designing our levels, there's nothing stopping us from creating superb games with this system.

Vertical scrollers

Typical example: Xenon II

The game is set in a large vertical playfield which scrolls smoothly down the screen. The playfield can be generated using either the AMOS MAP definer or AMOS TOME. It can be scrolled with a range of simple programming techniques which I'll be discussing in Chapter 7.

Horizontal scrollers

Typical examples: Silkworm/Defender

Horizontal scrolling is perfect for landscapes, and has now become a common feature in many games. On the minus side though, it's decidedly slower than the equivalent vertical system. So although horizontal scrollers look prettier, vertical scrollers usually generate the smoothest action.

Complex scrollers

Typical example: Blood-money/Jetstrike

Complex scrollers stretch our Amiga to its absolute limits. The game is set in a large two dimensional playfield which scrolls smoothly in any direction. As a rule, it's easiest to scroll our playing area in one just direction at a time. Be warned! If it's necessary to change the direction continually, we'll need to indulge in a fair amount of extra programming. Sometimes, this will pay off handsomely, but on other occasions we'll be forced to expend a great deal of time and effort on the scrolling operations without any significant improvement in the overall gameplay. Having said that, we only need to look at Aaron Fothergill's excellent Jetstrike program to see what can be achieved. This was written almost entirely in AMOS Basic, and provides a terrific demonstration of complex scrolling in action.

When you are starting out, it's wise to limit yourself to the simpler static games. These are perfect for the beginner, as they provide a gentle introduction to Arcade games without causing any unnecessary headaches along the way. Once you've created one of these games and gained a little confidence, you will then be able to expand your program by adding some extra animation to the playing area. Believe me, this is MUCH easier than attempting to write a full-blown screen scrolling arcade game from scratch!

Screen scrolling is an art, and until you get the hang of it, it's a pretty arcane one at that! Rather than clog up this chapter with a mass of scrolling theory, I've placed all the scrolling stuff into its own separate chapter. Chapter 7 includes a detailed, step by step guide to scrolling anything from a simple message, to an vast alien landscape. For the moment however, I'll take some of my own advice, and concentrate on the simpler static games instead. When it's time to expand these programs, Chapter 7 will come into its own.

3.2.2 Choosing the scenario

As you may have noticed, all the most successful arcade games are centred around a single underlying theme. Blood-Money for instance, is set in a vast alien safari-park. The instructions for these games often include a detailed scenario which provides a crude justification for zapping the various aliens. Nobody expects the scenario to be particularly believable of course! That's not what it's for! After all, we already know the reasons for playing an arcade game perfectly well.

Aim of the game: Shoot the alien scumbags!

Justification: It's fun!

Obvious isn't it? The point about the scenario though, is that it establishes a consistent pattern to the design of the game screens. Each game now has its own particular atmosphere. So the attackers actually represent something, rather than being just attractive blobs of colour.

Here are a few scenario ideas to get you started. Where possible, I've included the locations of some appropriate image files on the AMOS Extras disk. Remember, these images can be freely used in any of your own games, so it's easy to start experimenting straightaway!

Catacombs

Sprite_600/Fantasy

Why not set the game in a mysterious graveyard or burial chamber? There would be plenty of scope to create a variety of ghouls and ghosts as our monsters, and there's certainly no shortage of source material. Scare the life out of your players!

Araknoid

We could also populate our game world with a variety of the most unpleasant insects we can think of. Imagine a simple house-fly blown up to giant proportions. Now that's my idea of a real bug eyed monster!

Battle of the Skies

Sprite_600/Flight

Sprite_600/Ground

Give the player the demanding job of clearing the skies of a range of advanced fighter aircraft or tanks. One demonstration of the idea can be found in the classic Silkworm program. But the same concept can provide the inspiration for hundreds of possible scenarios.

Warship

Sprite_600/Water

We could also assign the player to a warship fighting off hordes of attacking planes and ships. This would be perfect for a screen scrolling game, and it's not been exploited in too many games. Yet!

Underwater Encounter

Sprite_600/Water

The oceans support an incredible variety of alien lifeforms, from a deadly shark to a giant octopus. So we've the perfect excuse for a little creative mayhem. Give the players a submarine and ask them to navigate though the infested waters. Yes I know it's been used before, but if we take the trouble, we can easily think of dozens of exciting variations.

Land of the Dinosaurs

Send the player back through time to encounter some of the most terrifying monsters ever to have existed on this planet.

War of the Robots

Imagine a world overrun by robots of all shapes and sizes. Suppose the robots decided to destroy their creators. Asimov called it the Frankenstein syndrome, and made a fortune out proving it could never happen. It may be great SF, but the future could be very different! Give the player the job of single-handedly crushing the rebellion, and see what happens. You could have the makings of a terrific game!

Space Wars

Sprite_600/Aliens

Sprite_600/Space

Old hat! But very,very easy. The idea may have been done to death, but there's plenty of mileage in it yet! There's also the fact that the AMOS Extras disk contains dozens of ready

made aliens for us to use in our games. Look in the ALIENS and SPACE directories of the SPRITE_600 folder for some examples.

That's just a selection of the scenario possibilities. Have a bash at thinking up some of your own. It's fun!

3.2.3 Designing the attackers

Once we've decided on a general theme for our game, we're ready to design the various attackers. These can be created with either the AMOS Sprite Editor, or from an external drawing package such as Deluxe Paint. It's up to you! While we're drawing our images, it's vital to ensure that all our screen objects use exactly the same set of colours. This is especially important if we intend to create our objects using a number of different drawing packages. There are two main possibilities:

- If we need to load the colours from a IFF screen file into the AMOS Sprite Editor, we can use the Sprite Grabber utility from the AMOS Program disk. This allows us to grab any section of an IFF screen, and save it onto disk in the form of a standard sprite image. We can then load this image straight into the AMOS Sprite Editor along with all our original colours.
- Alternatively, we can easily save the colours used by our existing sprite images into a separate screen file:

```
Get Sprite Palette
Save Iff "Colours.IFF"
```

These instructions create a blank IFF screen containing our entire colour palette, which can then be subsequently loaded into practically any drawing package on the Amiga. We can now draw our images confident in the knowledge that we are using exactly the same colours as those in the original sprite bank.

Level guardians

The best arcade games also include impressive monsters at the end of each level. These level guardians can add a delightful extra challenge to our games, and will often require the use of special skills or equipment before they can be defeated. This gives us plenty of scope for indulging our imaginations, and allows us to have endless fun designing weird and wonderful gadgets for our player ship.

Normally, level guardians are animated separately from the rest of the program. So we can happily incorporate any extra features we like, without adding unnecessary complications to our animation routines. The thing to remember when we are generating these objects, is that we'll need to ensure that the width of each monster divides exactly by 16. This lets us implement each guardian as a single massive Blitter object, and animate it using all our favourite AMOS commands (including AMAL).

The only restriction is the amount of available memory. This can be minimised by using the Sprite Squash utility. Sprite Squash allows us to compact our images into a fraction of their usual space. It consists of two programs, both of which can be found on the AMOS Program disk. Don't worry if you can't find them on your own discs, as they are a fairly recent development. Copies are available from the AMOS PD library for a nominal handling charge.

Squash_A_Bob.AMOS

This program compresses our chosen images from the sprite bank and saves them into a separate file in the disk.

Squash_Procs.AMOS

Takes the compacted images and converts them back into Blitter objects. Full details of these routines can be found on the AMOS Extras manual or on a documentation file on the PD disk.

The AMOS sprite editor may be great for small images, but it's not really suitable for larger objects such as level guardians. It's therefore a good idea to draw all our guardians using an external drawing package such as Deluxe Paint. That's the approach used by the vast majority of professional games designers. Once we've drawn our images, we can then import them directly into AMOS Basic using the Sprite Grabber utility as before.

3.2.4 Designing the player's ship

The design of the player's ship is vitally important, as this forms the centrepiece of the entire game. Obviously, the player can actually be controlling anything from a microbe to an aeroplane. From now on, I'll use terms *Ship* and *Aliens* to represent the player and the attackers respectively.

Nowadays, there's a trend towards large ships with lots of optional extras. This allows us to gradually increase the strength of the alien attackers during the course of a game, without completely overwhelming the unsuspecting player. Enhancements can be awarded either as a special bonus, or can be *purchased* as in Blood-money. In this case, cash is awarded for every complete wave destroyed by the player, and there's a *shop* at the end of each level.

Here is a brief list of the types of enhancements we can incorporate in our AMOS Basic games:

Shields: Simple! They just increase the number of hits which can be taken before the player loses a life.

Lasers: Lasers fire concentrated beams of energy at the opposing ships, destroying them in a single blast. Just to make life harder, there's often a limit to the length of time the laser can be fired in a continuous burst. When this limit is exceeded, the lasers are deactivated for a few seconds for recharging purposes, and the player is left practically defenceless. This stops players with auto-fire joysticks from effortlessly annihilating everything in their path, and adds a useful extra challenge to the game.

Rear-shots: Extra guns which bolt onto the back of the player's ship, and fire from the rear, destroying any baddies which attempt to creep up from behind.

Side-shots: As their name suggests, Side-Shots allow the player to fire missiles from the sides of the ship. They are often provided in screen scrolling games to scour the walls of alien gun emplacements.

Mines: Small bombs which can be dropped from the bottom of the player's ship. They can be released either by pressing a special key (space is favourite), or automatically whenever the player hits the fire button.

Time-bombs: Similar to mines, except that they explode violently after a couple of seconds, destroying dozens of our enemies in a single strike. If we're feeling nasty, we could insist that the player gets well out of range before they explode. So unless the players are really careful, the bomb could detonate right in their faces!

Energy bombs: Destroy all the aliens on the screen in one massive explosion. They are ideal for getting the player out of a really tight corner, and make great end of level bonuses.

Guided missiles: Crash straight into the nearest alien destroying it utterly. If the alien is moving, the missile will follow it relentlessly on the screen. Feel free to give these missiles to the attackers as well!

Teleports: The teleport or hyperspace option instantly transports the player's ship to a new (random) position on the screen. If this new position is already occupied by an alien, the ship will usually be destroyed and the player will lose a life, so it should only be used as a last resort. It's dangerous!

Outriggers: Separate ships which automatically move in step with the player's ship. Whenever the player moves the ship, or fires a missile, the outriggers respond too. This results in a dramatic increase to the overall fire-power available. So it's the perfect time to raise the stakes, and provide our aliens with a few welcome reinforcements!

It's easiest to draw up a separate image for each possible combination of outrigger and ship. This allows us to assign a single bob for both objects, and simplifies our control system. We can now use the same joystick routine to handle any number of objects simultaneously. All we need is a new procedure to handle the extra missiles.

Drones: Identical to outriggers except that they are completely independent of the players ship. Drones are controlled directly from the computer, and supply valuable assistance to the player, at a price. Although drones are a little fiddly to generate, they are well worth the effort, especially in a screen scrolling game.

3.3 Components of a Shoot-'em-up

After we've invented a scenario, and produced a specification, it's time to begin work on our actual program design. The starting point is to make a detailed list of all the activities which will need to be performed by the program. These will form the basis of the subroutines we'll be using in our final game.

As an example, take your favourite arcade game, and try to name some of the main program sections. Don't worry too much about how these routines might be created for the moment. I'll be discussing them in detail later on. Hopefully, you will eventually be left with a list which looks rather like this:

Set up the screen
Load up the attack wave
Move the aliens
Control the player
Fire the player's missile
Fire the alien's missiles
Detect the collisions
Handle any special objects (Lasers/Drones/Outriggers etc..)
Animate the background.

3.3.1 Anatomy of an arcade game

The next stage in the development process is to produce a detailed description of the mechanics of our game. This can be written using the pseudo-code notation I introduced in Chapter 2. If you haven't already familiarised yourself with this system, it's well worth making a brief detour before continuing. It's really just a specialised form of normal English. Anyway, here's one I made a little earlier:

Figure 3.1 Game plan of an arcade game

```
Set up background screen
Repeat
Load attack wave
Set up attack wave
  Repeat
  Read Joystick or Mouse
  Detect for a collision between player and a missile
  Detect for a collision between player and attacker
  Move player
  Move attacking ships

  If player is hit
    Then
      Destroy player
      Lives=Lives-1
    Endif

  If fire pressed
    Then
      Fire missile
    Endif
  Detect collision between player's missile and enemy

  If enemy hit
    Then
      Destroy enemy
      Attackers=Attackers-1
      Increase score
    Endif
  Move player's missile
  Handle any special objects (Drones/Outriggers/Lasers etc..)

  If alien fires
    Then
      Choose one of the attacking ships
      If there's a free missile
      Then
        Fire missile from ship
      Endif
    Endif
  Move attackers missiles
  Animate background screen
  Until Lives=0 or Attackers=0
Until Lives=0
Another game?
```

I'll now take each of these activities in turn, and discuss how they might be performed in one of our AMOS Basic programs.

3.3.2 Screen initialisation

At the beginning of our game, we'll obviously need to define our screens and load up our various graphics. This can be accomplished by either loading a previously created IFF picture, or displaying a MAP generated by the AMOS Map Definer (see Chapter 7). Additionally, if we are intending to use Blitter objects in our game, we'll also need to ensure that the new program screen uses *double buffering*. The minimum practical set up is therefore something like this:

```
Rem Turn off flashing text cursor, and clear the screen
Curs Off:Flash off:Cls 0
Rem This can be omitted if you're just using sprites (Rare)
Double buffer
Rem load an optional background screen
Rem The colours should be the SAME as those used in your bob images
Load IFF "Picture.Iff"
Load "Images.abk":Rem Load optional sprite file
Get Sprite Palette
```

The DOUBLE BUFFER command is VITAL. If we attempt to use the Bob commands without first activating the Double Buffering system, all our Blitter objects will flicker uncontrollably whenever they are moved. Ugh!

Note that since sprite/bob images are saved automatically along with our programs, we can easily install them permanently in memory. All we have to do is load them into AMOS Basic and save our program onto the disk in the normal way. They'll now be ready for use at a moments notice.

During development though, it's probably better to keep the images in a separate file, as this makes them much easier to change during the evolution of our game. It also keeps the size of our resulting program files down to an absolute minimum. It's well worth getting into the habit of removing our images from memory before saving our programs onto the disk. This can be accomplished by simply typing:

```
erase 1
```

from direct mode. Bingo! We've just stripped 10-50k from our program files! In practice, the memory savings can quickly mount up, especially if we make frequent backups of all our programs on several different discs (**Recommended!**)

The final part of our initialisation procedure is to load up our image colours from the sprite bank using:

```
Get Sprite Palette
```

Try removing it from one of my example programs and see what happens! It's not exactly pleasant is it?

3.4 Controlling the Player's Ship

I'll now demonstrate a number of techniques which can be used to control the player's ship. Each approach has its own combination of strengths and weaknesses, so the final choice is obviously entirely up to you.

Before we continue, it's worth saying a few words about the *hot spot* of an object. This is the point on the bob from which all its screen coordinates are measured. The hot spot is rather like a *handle* which can be used to drag the image around the screen. Figure 3.2 may make things a little clearer.

The position of the hot spot can be changed at any time using the HOT SPOT command from AMOS Basic. As a default, it's set to the top left hand corner of the image, but it can actually be placed almost anywhere. In many cases, it's better to move the hot spot to the centre of the image, as this makes it much easier to position our objects on the screen. The change can be made with the following command:

```
Hot Spot image,$11
```

where *image* is the number of the image we wish to adjust in the AMOS sprite bank. Note that the new position of the hot spot will be automatically saved along with our sprite images.

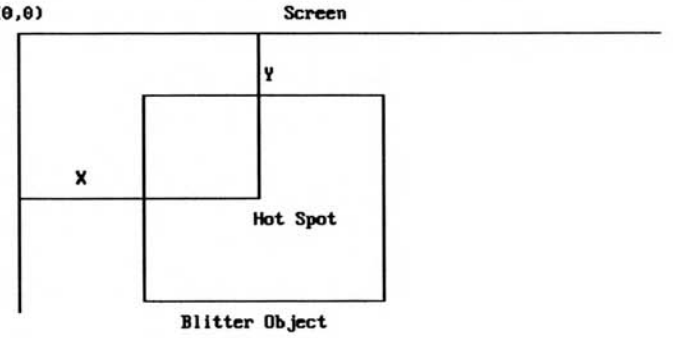

Figure 3.2 The HOT SPOT

If you want to experiment a bit, there's also a more general form of the HOT SPOT instruction:

```
Hot Spot image,x,y
```

x,y are the distances of the hot spot from the top left corner of the image. Try using negative values to position the hotspot outside the current image, for some really weird results.

3.4.1 Controlling the ship with the mouse

The mouse provides us with an ideal way of entering our commands and moving the ship smoothly around the screen. The general technique is extremely simple. The first stage is to remove the existing mouse pointer from the screen with HIDE, so it doesn't interfere with our ship. We can now read its position directly using the AMOS Basic X/Y MOUSE functions.

hx=X MOUSE

hy=Y MOUSE

These return the current hardware coordinates of our mouse on the screen. As you may have gathered, these functions will still work fine even if the mouse pointer has been removed completely with HIDE. So there's absolutely no need for the pointer to be visible.

The only (minor) complication with this system, is if we're using a Blitter object (bob). Since the mouse only works with hardware coordinates, we'll need to convert these values into the equivalent screen versions using the X SCREEN and Y SCREEN functions.

```
Rem Controlling the ship with the mouse
Rem AMOS Basic version
Hide:Rem remove mouse pointer from the screen
Rem read the position of the mouse
X=X Screen(X Mouse):rem Gets the X coordinate the Invisible mouse
Y=Y Screen(Y Mouse):rem Gets the Y coordinate of the mouse
Rem Move the ship
Rem SHIP is the number of bob your using for your ship
Rem IMAGE is an image from the current sprite bank
Bob SHIP,X,Y,IMAGE
```

This should be placed in the main loop of our game.

Now for an AMAL version of the same routine. Since AMAL commands are performed 50 times a second using interrupts, we only need to call this routine once. After that, our ship will be moved automatically by the AMAL system, just like the original mouse pointer. AMAL includes two useful functions which mimic the AMOS X and Y MOUSE commands exactly:

=XM

Gets the X coordinate of the mouse (in hardware coordinates).

=YM

Returns current Y coordinate of the mouse.

XM and YM are standard AMOS functions and can be assigned to an AMAL register with Let , or tested using the If command as appropriate.

Example 3.1: Controlling the ship with the mouse

```
Rem AMAL version
Rem XS and YS are AMAL versions of X SCREEN and Y SCREEN
Rem The format's more or less identical
MB$="M: Pause; Let X=XS(0,XM); Let Y=YS(0,YM); Jump M"
Rem That's it!
Rem if you are using a sprite instead of a bob, replace with:
Rem MS$="M: Pause; Let X=XS; Let Y=YS; Jump M"
Rem Load some images and test
Load "Extras:/sprite_600/Aliens/Alien4.Abk"
rem Assign an animation channel to the ship
Channel 1 to Bob 1
Rem Position the Bob
Bob 1,160,100,1
Amal 1,MB$:Amal On
Rem repeat forever!
Do
Loop
```

If we want to move our ship over just a part of the playing area, we can optionally restrict the mouse movements with the LIMIT MOUSE command:

Limit Mouse TX,TY To BX,BY

LIMIT MOUSE generates a rectangular *cage* around the selected screen area. The mouse is now trapped, and AMOS will ignore any attempt to move it outside the new boundaries. TX,TY hold the hardware coordinates of the top left hand corner of the movement zone, and BX,BY the coordinates of the point directly opposite. If you find hardware coordinates a little too complicated, don't forget that you can convert them from the equivalent screen versions using the XHARD and Y HARD functions :

```
Limit Mouse X Hard(0),Y Hard(0) To X Hard(160),Y Hard(100)
```

This limits the mouse to screen coordinates from 0,0 to 160,100

3.4.2 Controlling the ship with the joystick

The mouse may be fine for some situations, but most games also allow us move the player using the joystick.

The Joystick functions

AMOS Basic provides us with six built-in functions for just this purpose. These make it very easy to read the precise position of the joystick at any time in our program.

=Jleft(S)

=Jright(S)

=Jup(S)

=Jdown(S)

These functions test a joystick plugged in socket S for a movement in the appropriate direction. If the test is successful, the functions will return a value of TRUE (or -1), otherwise we'll get a value of FALSE (or 0). S is the number of the joystick socket at the back of your Amiga and ranges from 0 to 1. Generally, the mouse will be placed in socket zero and the Joystick in socket 1.

=Fire(S)

This checks the fire button in exactly the same way. A value of TRUE (-1) indicates that the trigger has been pressed, otherwise a value of FALSE(0) will be returned.

=Joy(S)

The JOY function is slightly more complex, as it returns a bitmap containing a binary bit for each possible movement direction. If the joystick has been pulled in this direction, the relevant binary bit will be set to 1. Technically minded readers adore binary notation, as it's delightfully exact, but personally I tend to avoid it like the plague! So here's a complete list of all the values which can be returned by the AMOS JOY function.

Normal	**With Fire Button pressed**
1	17
5 9	21 25
4 0 8	20 16 24
6 10	22 26
2	18

The numbers around the circle indicate the results of moving the stick in the appropriate direction, and the value at the centre is the default. If, for instance, we were to move the joystick left, JOY would return a value of 4 or 20, depending on whether we'd pressed the fire button as well.

Although the JOY function is quite powerful, it's decidedly fiddly to use. So it's actually no joy! It does however, provide a simple way of checking whether the player has moved the joystick. If the value returned by Joy(1) is greater than zero, we know immediately that the joystick has been manipulated in some way. We can then check the various directions individually with the appropriate JLEFT, JRIGHT, JUP, and JDOWN functions.

Each movement direction can be tested using a separate IF..THEN statement. This can be seen from the following bit of pseudo-code:

```
If Jleft(1) Then Move left
If Jright(1) Then Move right
If Jup(1) Then Move Up
If Jdown(1) Then Move Down
```

The Move commands are fairly simple, as we can move the ship along the screen directly with a BOB or SPRITE command. Remember, the position of our object will always be stored as a simple pair of X,Y coordinates. The X coordinate holds the horizontal position, and the Y coordinate contains its vertical component.

At the start, the coordinates of our object can be saved in a couple of AMOS variables such as SX and SY. We could now move our ship one place to the left using:

```
SX=SX-1
Bob S,SX,SY,SHIP
```

where *S* is the bob number we've assigned to our ship, and *SHIP* holds the image number we'll be using from the Sprite bank. Similarly,

```
SX=SX+1 : Bob S,SX,SY,SHIP: Rem Moves the ship Right
SY=SY-1 : Bob S,SX,SY,SHIP: Rem Moves the ship Up
SY=SY+1 : Bob S,SX,SY,SHIP: Rem Moves the ship Down
```

We can incorporate these equations into our joystick tests like so:

```
Rem SX,SY hold the coordinates of our ship
Rem If joystick pulled left the move ship left
If Jleft(1) Then SX=SX-1
Rem  Check for a Right and Increase SX by 1
If Jright(1) Then SX=SX+1
Rem If player moves UP reduce SY by 1
If Jup(1)  Then SY=SY-1
Rem Move down
If Jdown(1) Then SY=SY+1
Bob S,SX,SY,SHIP:Rem Reposition Bob S to a new coordinates
```

Unfortunately, we've forgotten something! What will happen if the player attempts to move the ship completely off the screen? With our present system, SX and SY can be easily set to any value we like, even if it's outside the current playing area. This is a serious mistake! Before we move our ship, we obviously need to check to see if the chosen direction makes sense. So we'll need to add an additional test to each If..Then statement in our program.

```
Rem If Player pulls joystick left
Rem and the ship is not already at the left hand edge
Rem move ship
If Jleft(1) and SX>0 Then SX=SX-1
Rem If the joystick is moved right
Rem and SX is less that the current screen width
Rem move ship
If Jright(1) and SX<320 Then SX=SX+1
Rem Test for a legal move upwards
If Jup(1) and SY>0 Then SY=SY-1
Rem Check for a legal move down
If Jdown(1) and SY<199 Then SY=SY+1
Rem Reposition Bob S at new screen coordinates
Bob S,SX,SY,SHIP
```

Here's a complete example for you to play around with:

Example 3.2: Controlling the ship with the joystick

```
Load "Extras:Sprite_600/space/ship3.abk"
Double buffer
Flash off:Get Sprite Palette
Cls 0
rem SX,SY are the coordinates of the players ship on the screen.
rem SHIP is the image number we've assigned to our ship
SX=160:SY=100:SHIP=1
Do
    If Jleft(1) and SX>0 Then SX=SX-1
    If Jright(1) and SX<320 Then SX=SX+1
    If Jup(1) and SY>0 Then SY=SY-1
    If Jdown(1) and SY<199 Then SY=SY+1
    Bob 1,SX,SY,SHIP
    Wait Vbl:rem Makes motion EXTRA smooth!
Loop
```

(I'll be explaining the mysterious WAIT VBL command a little later.)

An AMAL joystick handler

Another idea, is to write our joystick routine entirely using the AMAL animation system. The advantage of the system is that once we've set up the AMAL command string, we can forget about it completely! The ship will now behave just like the mouse pointer, and will be moved automatically around the screen without any further effort on the part of our program.

Let's have a look at a small AMAL joystick handler. This is similar to the one on the AMOS Extras disk, but I've been expanded it a bit to make it more usable.

```
Rem Simple AMAL joystick handler
Rem NOTE: I've spaced it out so it's easy to read. Don't type the
Rem blank lines or these REM statements!
Rem First set up the sensitivity of the joystick
Rem RX=Distance moved when the stick is pulled left or right
Rem RY=Distance moved when the stick is pulled up or down
A$="Let RX=2; Let RY=2 ; "
Rem Next define the coordinates of a rectangular box which will
Rem be used to contain the ship (Similar to Limit Mouse)
Rem RL=X Coordinate of the top left hand corner of the box
Rem RR=X Coordinate of the bottom right corner
Rem RD=Y Coordinate of top left
Rem RU=Y Coordinate of bottom right
Rem The precise values depend on the requirements of your program
Rem and the position of the hot spot of your image
Rem Experiment!
Rem If you're using SPRITES, add 128 to RL and RR
Rem and 50 to RU and RD to convert into hardware coordinates
A$=A$+"Let RL=16; Let RR=304; Let RU=100; Let RD=188;"
Rem Now test each possibility and jump to the appropriate routine
Rem J1 is equivalent to JOY(1) in AMOS Basic
A$=A$+"L: If J1&1 then Jump E ;" : Rem up
A$=A$+"  If J1&2 then Jump F ;" : Rem down
A$=A$+"  If J1&4 then Jump G ;" : Rem left
A$=A$+"  If J1&8 then Jump H ;" : Rem right
Rem Jump back to start
A$=A$+"Jump L;"
Rem UP
Rem Y holds the Y coordinate of the current object
```

```
Rem when it's changed, AMAL moves the object automatically
Rem If the object is already at the top of the screen
A$=A$+"E: Let Y=Y-RY ; If Y>RU then Jump L;"
Rem This line reverses the movement
A$=A$+"  Let Y=Y+RY; Jump L;"
Rem DOWN
Rem Move object RY units down
Rem Check for an illegal move
A$=A$+"F: Let Y=Y+RY ; If Y<RD then Jump L;"
A$=A$+"  Let Y=Y-RY; Jump L;":Rem Reverse if illegal
Rem LEFT
Rem AMAL variable X holds the current X coordinate of our object
Rem If it's changed, the object will be moved across the screen
Rem Move object RX units left. If move is not allowed, then reverse
A$=A$+"G: Let X=X-RX ; If X>RL then Jump L;"
Rem Here!
A$=A$+"  Let X=X+RX; Jump L;"
Rem RIGHT
Rem Move object RX units right
A$=A$+"H: Let X=X+RX ; If X<RR then Jump L;"
Rem Move object back to the left if it moves past the right hand edge
A$=A$+"  Let X=X-RX; Jump L;"
Rem Example program
Rem Initialisation
Load "extras:sprite_600/space/ship2.abk"
Curs Off : Flash Off : Hide
Get Sprite Palette
Double Buffer
Cls 0
Rem Set variables
S=1 : SHIP=2
Rem Centre hot spot
Hot Spot SHIP,$11
Rem place Bob S at the centre of the screen
Bob S,160,100,SHIP
Rem Assign and AMAL program to Bob S
Channel 1 To Bob S
Rem Activate program
Amal 1,A$ : Amal On
Rem Draw a box around the screen
Box 0,0 To 319,199
Rem Wait...
Do
Loop
```

This program may look pretty complicated, but it's really just an AMAL version of my original AMOS Joystick handler. The joystick is tested using a series of AMAL If statements, and each movement is carefully checked to see if it lies inside the screen area defined by the zone RL,RU to RR,RD.

Note the use of the AMAL If command. This is actually quite different from the AMOS version, so it's easy to get confused. Since we'll be using it later, a quick explanation is probably in order. The AMAL If instruction has the format:

```
If test Jump Label
```

where *test* is a single test such as Y<200, X>0. The test will be checked from left to right, and cannot include AND,OR or brackets (). Also note that in the program the *then* is in lower case and will be completely ignored by the AMAL system. It's only there to make the program more readable.

Label is the name of a AMAL label we've defined elsewhere in our animation string. If the test is successful, the Jump command will restart the AMAL program at the instruction immediately after Label. Unlike the equivalent AMOS command, we're only allowed to use a Jump in this position. So we can't execute other AMAL instructions such as Let or Move instead.

```
If Y<200 then Let Y=0
If X<0 then Move 100,100,1
```

These will generate an illegal function call ERROR! This has a dramatic effect on the structure of our AMAL programs. Take the joystick right routine for instance:

```
H: Let X=X+RX ; If X<RR then Jump L;
 Let X=X-RX; Jump L;"
```

In AMOS Basic, I'd have written it on one line:

```
H: If X<RR-RX Then X=X+RX
```

But because the AMAL version of If can only perform Jumps, I've been forced to add the value of *RX* to *X* before the test.

```
H: Let X=X+RX;
```

If the result lies outside the permitted area, RX then needs to be laboriously restored to its original value with Let. This makes the final program look insanely difficult, even though I'm really doing something very simple.

```
If X<RR then Jump L;
Let X=X-RX; Jump L;"
```

As you can see, If requires us to indulge in the occasional convoluted bit of programming. The AMAL If command has one stunning advantage over its AMOS equivalent: it's very very fast! So once you've learned the ropes, it's an incredibly useful instruction.

On the face of it, AMAL is ideal for our Joystick handler. All it takes is a simple set up routine, and the ship practically moves itself. Before you get carried away however, it's worth noting a word or two of caution. Since AMAL is performing all the movement automatically, we've absolutely no control over when the ship is actually moved on the screen. This makes life very difficult when we are trying to detect collisions, as the ship can often be moved a considerable distance between a collision occurring on the screen, and its successful detection. By the time our program finally catches up, the ship might be somewhere totally safe. Imagine the look on the player's face when we suddenly kill him off for no apparent reason!

Thankfully, the problem is fairly easy to solve. Whenever we move an object on the screen, it's usually redrawn at the new position more or less immediately. We can however, turn this automatic screen updating with the UPDATE OFF command, and display the objects ourselves using UPDATE. We can now UPDATE the screen just before the test for a collision. This will ensure that the collision will be detected the moment it happens on the screen. If this seems hopelessly involved, then rest assured, I'll be discussing this system in detail in Section 3.7.3.

3.4.3 Universal control system

Finally, here's a Universal control system (UCS) for use in one of your own games. It automatically reads the mouse or the joystick and moves your ship on the screen accordingly. For convenience sake, I've placed it in a little subroutine, but it could obviously be incorporated directly into your games main loop.

```
UCS:

   Rem Use the Joy function to see if the joystick has been moved
   If Joy(1)
       Rem Check each direction in turn
       Rem DX and DY set the size of each movement step
       Rem This ship is limited to the rectangular area
       Rem defined by TX,TY to BX,BY
       If Jleft(1) and(SX>TX)
         Add SX,-DX
       End If
       If Jright(1) and(SX<BX)
         Add SX,DX
       End If
       If Jup(1) and(SY>TY)
         Add SY,-DY
       End If
       If Jdown(1) and(SY<BY)
         Add SY,DY
       End If
       Rem Move the mouse pointer to the new position
       Rem Bet you didn't think this was allowed
       X Mouse=X Hard(SX)
       Y Mouse=Y Hard(SY)
   Else
       Rem Read the mouse position
       Rem add LIMIT MOUSE command to restrict the mouse
       Rem to the movement area defined by TX,TY to BX,BY
       SX=X Screen(X Mouse)
       SY=Y Screen(Y Mouse)
   End If
   Rem Move the ship
   Bob 1,SX,SY,SHIP
Return
```

The reason for the Universal in UCS is because it's capable of moving the ship in any direction, at any speed, and at any screen position, using either the mouse or the joystick. Hopefully, it should be reasonably familiar, as it uses a similar system to the one in Example 3.2.

The ship is limited to a rectangular box defined by the coordinates TX,TY to BX,BY. Each tug of the joystick moves the ship by a set amount. This depends on the current direction, and is specified by the variables DX,DY. DX sets the sensitivity of the horizontal movements, and DY the speed of the vertical ones.

The clever stuff is in the way I've managed to read both the mouse and the joystick simultaneously. This exploits a trick of the X MOUSE and Y MOUSE functions. Normally, X MOUSE and Y MOUSE are used to return the position of the mouse on the screen. But as you can see from the UCS, they are also capable of MOVING the mouse pointer as well. This allows my joystick handler to move the mouse pointer straight to the new SHIP coordinates in SX,SY. So the next time the mouse is read, the X MOUSE and Y MOUSE functions will hold the correct position of the ship on the screen.

The neat thing about this system, is that it really does allow us to use either the mouse or the joystick at will. The thorn in the flesh though, is that there's absolutely no control over the sensitivity of the mouse movements. So DX, and DY only apply to the joystick. Well, we can't have everything, I suppose! Here's an example:

```
Rem Demonstrates joystick/mouse controller
Rem load some sprites from the EXTRA's disk
Rem place this in any drive!
Load "extras:Sprite_600/space/ship1.abk"
Rem Initialise the screen
Double Buffer
Flash Off
Get Sprite Palette
Curs Off:Cls 0
Rem Make up a simple starfield
Gosub STARS
Rem Set the sprite image used by the player's ship
SHIP=19
Rem move the hot spot to the centre of the ship
Hot Spot SHIP,$11
Hide : Rem Remove mouse pointer from the screen
Rem Set distance moved on the screen by each tug of the joystick
rem DX holds the size of a JLEFT or JRIGHT movement
Rem DY = the vertical distance when you move the joystick up or down
DX=8 : DY=8
Rem SH and SW hold the width and height of the SHIP image respectively
SH=32 : SW=32
Rem TX,TY sets coordinates of the top left corner of movement window
Rem BX,BY specify the coordinates of the bottom right corner
Rem Try using your own values instead
TX=0 : TY=128 : BX=320 : BY=192
Box TX,TY To BX-1,BY-1:Rem Draw a box around the movement area
Rem SX,SY hold the current SCREEN coordinates of the player's ship
SX=TX : SY=TY
Rem Since I've placed the hot spot of the SHIP at the centre
Rem I need to adjust the size of the movement area slightly
Rem This can be avoided by removing the previous HOT SPOT command
Rem but this complicates the missile routines I'll be showing
Rem you later. You can't win!
Rem Feel free to experiment!
TX=TX+SW/2 : TY=TY+SH/2 : BX=BX-SW/2 : BY=BY-SH/2
Rem Limit the mouse to the selected movement area
Limit Mouse X Hard(TX),Y Hard(TY) To X Hard(BX),Y Hard(BY)
Rem Main program loop
Do
  Gosub UCS:Rem Call the Universal Control System
  Wait Vbl :Rem Wait for a 1/50 of a second
Loop
Stop
STARS:
Rem draw one hundred pretty stars on the screen
Autoback 0
For I=1 To 100
    Ink Rnd(15)
    Plot Rnd(320),Rnd(200)
Next I
Screen Copy Logic To Physic
Autoback 2
Return
Rem Universal control system
UCS:
   Rem Test Joystick
   If Joy(1)
        If Jleft(1) and(SX>TX) : Add SX,-DX : End If
        If Jright(1) and(SX<BX) : Add SX,DX : End If
```

```
        If Jup(1) and(SY>TY) : Add SY,-DY : End If
        If Jdown(1) and(SY<BY) : Add SY,DY : End If
        Rem Move mouse to new position
        X Mouse=X Hard(SX) : Y Mouse=Y Hard(SY)
   Else
        Rem Test mouse
        SX=X Screen(X Mouse) : SY=Y Screen(Y Mouse)
   End If
   Rem Redraw SHIP
   Bob 1,SX,SY,SHIP
Return
```

3.5 Missiles

Now we've set up the player's ship, it's time to deal with its associated fire-power. Moving the missiles is easy, as we can use the simple AMAL Move command I showed in Chapter 1. Just to refresh your memory, here's a reminder of the general format:

`Move horizontal distance,vertical distance,n`

n holds the number of movement steps. The lower the value, the faster the movement.So:

```
Move 100,100,2
```

moves 100 units right and 100 units down in two steps. Each step is 100/2 or 50 units long. Move is only available from the AMAL animation system, and cannot be executed directly from AMOS Basic and must be loaded into a string and called up using the AMAL command:

```
Channel 1 To Bob Miss:Rem assign an AMAL animation channel to MISS
S$="Move 100,100,2":Rem Define movement string
Amal 1,S$: Rem Assign movement string to AMAL program number 1
Amal On 1: Rem Execute program 1
```

3.5.1 Firing a missile

Usually, missiles progress smoothly from the players ship towards the edges of the screen. So they can be implemented using commands like:

```
Move 0,-210,30  (For Static games and Vertical scrollers.)
Move 320,0,40  (Suitable for horizontal scrolling games.)
```

3.5.2 Moving the Missile

Alas, it's not quite that simple. We first need to position the missile at the top of the player's ship. In order to do this, we have to know it's present location on the screen. That's easy if we are controlling the ship from AMOS Basic, as the coordinates can be read directly from the appropriate variables (SX,SY). We could now move our missile into place with something like:

```
Bob 2,SX,SY-32,2
```

This moves a missile assigned to bob number 2 just above the player's ship. Note that I'm assuming we've already positioned the hot spot to the centre of the ship and missile images with HOT SPOT. Otherwise, we'd need to add extra values to the coordinates to centre the missile above the ship:

```
Bob 2,SX+8,SY-16,2
```

Unfortunately, if we're controlling the ship using AMAL, the above system won't just work, as

we've no way of knowing the current ship coordinates. The solution is to make use of a couple of AMOS Basic functions called X Bob and Y Bob. These return the present coordinates of any bob on the screen.

X=X Bob(bob_number)
Gets the X coordinate of a Blitter object

Y=Y Bob(bob_number)
Returns the Y coordinate of a Blitter object.

The missile can now be positioned using a line such as:

```
Bob MISS,X Bob(SHIP),Y Bob(SHIP)-32,MI
```

where *MISS* and *MI* are variables holding the bob number and the image number, respectively. Now for an example:

```
MISSILE:
  Rem The next line limits you to about 2 shots per second
  Rem Change according to taste!
  If Timer<TM+20 Then Return
  Rem MISS is the bob number of your missile
  Rem MI holds the image number
  Rem Position missile above ship (Bob number in SHIP)
  Bob MISS,X Bob(SHIP),Y Bob(SHIP)-32,MI
  Rem assign an amal animation string to missile
  Amal MISS,"Move 0,-210,30"
  Amal On MISS
  TM=Timer
Return
```

We'd also have to set up an AMAL animation channel with:

```
MISS=2:Rem for example
Channel MISS to Bob MISS
```

We could now fire the missile with:

```
If Fire(1) and Mouse Click Then Gosub MISSILE
```

So far, I've naturally assumed that we are going to move our missile with the AMAL animation system. However, if we are already using this to move or animate our aliens, we may well find ourselves running out of animation channels. As a default, AMAL only allows us to handle a maximum of 16 objects at once. Although we can remove the restriction with the SYNCHRO command (see later), it's often easier to fire our missiles directly from AMOS Basic. Here's a new version of our MISSILE routine which uses this idea.

```
MISSILE2:
  Rem The next line limits us to about 2 shots per second
  Rem Change according to taste!
  If Timer<TM+20 Then Return
  Rem MISS is the bob number of our missile
  Rem MI holds the image number
  Rem Position missile above ship (Bob number in SHIP)
  XB=X Bob(SHIP):YB=Y Bob(SHIP)-32
  Bob MISS,XB,YB,MI
  Rem load coordinates into a couple of missile variables
  MX=XB:MY=YB
  TM=Timer
Return
```

We can now move the missile by placing the following line inside our main program loop:

```
Add MY,-1: Bob MISS,MX,MY,MI
```

3.5.3 Multiple missiles

Obviously, few real games restrict the player to just a single solitary missile. But it's easily enough to adapt the previous routine to handle several missiles at a time. The simplest option is to assign several bobs for all the missiles and cycle through them one by one using the ADD command. Supposing we had decided to assign bob numbers 2 to 8 for our missiles. We'd first have to set up the various animation channels using the CHANNEL command like so:

```
For C=2 To 8
   Channel C to Bob C
Next C
```

We could then load the first and last missile numbers into a couple of variables:

```
MFIRST=2
MLAST=8
```

Finally, we'd set up a variable to hold the current missile number:

```
MISS=2
```

We could now fire off our missiles by placing using the following lines in our control routine:

```
If Fire(1) and Mouse Click
 Add MISS,1,MFIRST To MLAST
 Gosub MISSILE
Endif
```

The first time this routine was called, the missile would be assigned to bob number 3. The second missile would be then allocated to bob 4, and the sequence would continue:

Bob used for missile

5
6
7
8
2 Jumps back to the first bob number
3
4
5
:

MISSILE is the same routine I showed you earlier. Since we have set things up so carefully, it doesn't need any further changes. Hurray!

Whenever the player presses the fire button, a new missile will be immediately launched using the next bob in our list. If this bob is already assigned to a missile, it will be removed from the screen and released again from the top of the ship. The resulting effect may be slightly weird, but it's certainly usable. It's even found its way into several commercial games!

Example 3.3 Multiple missiles

```
MFIRST=2 : Rem First missile
MLAST=8 : Rem Last missile
SHIP=33 : Rem Ship image
MISSILE=49 : Rem Missile image
Load "df1:Sprite_600/space/ship1.abk" : Rem load the ship from the extras
disc
Load "df1:sprite_600/aliens/alien4.abk",1 : Rem tag the aliens onto the
end
Rem Set hot spot of ship to centre
Hot Spot 1,$11
Rem Set up screen
Double Buffer
Flash Off : Get Sprite Palette
Rem Make up a missile
MAKE_OBJECTS
Rem Assign channel 1 to the ship
Channel 1 To Bob 1
Rem Allocate some channels to the missiles
For C=MFIRST To MLAST
   Channel C To Bob C : Bob C,0,-100,1
Next C
Rem Create a simple starfield
Hide On : Curs Off : Cls 0 : Paper 0 : Pen 1
Centre "<A simple missile system>"
For I=1 To 100 : Plot Rnd(320),Rnd(200),Rnd(16) : Next I
Gosub JAMAL : Rem initialise AMAL joystick handler
MISS=MFIRST : Rem Set first missile
Rem Main loop
Do
   Rem Fire missile
   If Fire(1) Then Gosub MISSILE : Add MISS,1,MFIRST To MLAST
Loop
JAMAL:
Rem A simple AMAL Joystick handler
A$="Let RX=1; Let RY=1 ; "
A$=A$+"A: If J1&1 then Jump E;" : Rem up
A$=A$+"B: If J1&2 then Jump F;" : Rem down
A$=A$+"C: If J1&4 then Jump G;" : Rem left
A$=A$+"D: If J1&8 then Jump H;" : Rem right
A$=A$+"Jump A ;"
A$=A$+"E: Let Y=Y-RY ; If Y>0 then Jump A;"
A$=A$+"   Let Y=199; Jump A"
A$=A$+"F: Let Y=Y+RY ; If Y<199 then Jump A;"
A$=A$+"   Let Y=0; Jump A"
A$=A$+"G: Let X=X-RX ; If X>0 then Jump A;"
A$=A$+"   Let X=320; Jump A;"
A$=A$+"H: Let X=X+RX ; If X<320 Jump A;"
A$=A$+"   Let X=0; Jump A;"
Bob 1,160,100,1 : Rem position ship
Amal 1,A$ : Amal On : Rem Assign ship to joystick
Return
MISSILE:
Rem set delay between each volley
If Timer<TM+12 Then Return
Shoot
Bob MISS,X Bob(1),Y Bob(1)-5,MISSILE
Amal MISS,"Move 0,-210,60" : Amal On MISS
TM=Timer
```

```
Return
Procedure MAKE_OBJECTS
   Rem create missile
   Cls 0 : Ink 1 : Ellipse 7,6,1,6 : Paint 7,6
   Get Sprite Length(1),0,0 To 16,13
   Rem hot spot at tip of missile
   Hot Spot Length(1),$12
End Proc
```

Although this system certainly works, it's better to only re-use our missiles after they've disappeared off the edge of the screen. Providing we assign plenty of missiles to the ship, the player will seem to have a practically inexhaustible supply.

Admittedly, if the user is firing continually, the program will still occasionally run-out of ammunition. In this case, the gun will lock-up, and the ship will be left temporarily helpless until one of the existing missiles becomes available. Fortunately, this isn't really much of a problem, as it simply forces the player to think before firing. It also discourages the use of those awful auto-fire buttons.

The easiest way to learn about this system is to look at an example. Believe me, it's far easier than it sounds. I'll begin by introducing you to a new subroutine called ALLOCATE. ALLOCATE checks through all the available missiles, and returns a list of the ones which are currently unused. It's intended to be called from inside our fire-missile routine. This loads the number of missiles it needs into the variable *MN*, and calls ALLOCATE to get the required bob numbers.

Each missile is then carefully checked to see if it lies inside the visible screen area. ALLOCATE achieves this by just reading the position using the X Bob and Y Bob functions, and comparing these coordinates with the selected screen boundaries. If the missile fails this test then it's obviously not being used, and it can be safely assigned to the ship.

ALLOCATE is fairly intelligent, so if the requested number of missiles are available, ALL will be loaded with the original value of MN, otherwise it will be set to the maximum number of free missiles. The bob numbers associated with these missiles are saved in an array called PMISS, which needs to be defined at the start of our program. Now you know how ALLOCATE works, have a look at the actual routine. It's surprisingly simple!

```
ALLOCATE:
  Rem This expects the missile routine to place the number of
  rem missiles required in MN
  Rem After the routine has done its work, ALL will be loaded with
  Rem the number of free missiles
  ALL=0
  Rem Check though the missiles to find some which are unused
  For M=MFIRST To MLAST
  BX=X Bob(M) : BY=Y Bob(M):Rem Get the coordinates of the missile
  Rem Checks whether the missile has crossed the edge of the screen
  Rem I've kept it flexible to allow for side-shots and rear-shots
  Rem If your missiles can only move upwards, then you can omit
  Rem some of these tests for extra speed
  If BX<0 or BX>320 or BY<0 or BY>200
        PMISS(ALL)=M
        Inc ALL
  Endif
  Exit If ALL=MN
  Next M
  Return
```

Note that this system makes absolutely no assumptions as to the way our missiles are being moved on the screen. So it works equally well with any of our previous movement routines.

After ALLOCATE has completed its work, the available bob numbers for our missiles will be loaded into the PMISS array. This array should be set up at the start of our program like so:

```
Rem Set to a higher value if to loose off a larger salvo at a time
Dim PMISS(4)
```

Here's a new version of the original fire-missile routine which demonstrates this technique.

```
MISSILE3:
  If Timer<TM+12 Then Return
  Rem Find ONE free missile and place its number in PMISS(0)
  MN=1 : Gosub ALLOCATE
  Rem Sorry, no spare missile
  If ALL=0 Then Return
  Rem Start missile off above the ship
  Bob PMISS(0),X Bob(1),Y Bob(1)-5,49
  Rem Move missile using the AMAL Move command
  Amal PMISS(0),"Move 0,-210,60"
  Amal On PMISS(0) : Shoot
  TM=Timer
Return
```

If we run out of AMAL animation channels, we can also use the same ALLOCATE routine in conjunction with the direct missile system I showed you earlier.

Example:

```
MISSILE4:
  Rem Manual missile system (Doesn't use AMAL!)
  If Timer<TM+20 Then Return
  Rem Find ONE free missile and place its number in PMISS(0)
  MN=1 : Gosub ALLOCATE
  If ALL=0 Then Return
  Rem MISS is the bob number of your missile
  Rem MI holds the image number
  Rem Position missile above ship (Bob number in SHIP)
  XB=X Bob(SHIP):YB=Y bob(SHIP)-16
  Bob PMISS(0),XB,YB,MI
  Rem load coordinates into a couple of missile variables
  MX(PMISS(0)-MFIRST)=XB:MY(PMISS(0)-MFIRST)=YB
  TM=Timer
Return
```

Note that MX() and MY() are arrays which are assumed to have been defined previously during our program's initialisation phase. They're needed to hold a list of the current screen coordinates of our missiles.

We can now move these missiles directly in our program using a FOR..NEXT loop:

```
For M=MFIRST To MLAST
 Add MX(M),-1:Bob M,MX(M),MY(M),MI
Next M
```

Alternatively, if we are moving all our missiles with AMAL, we can speed up our allocation system by checking the AMAL animation channel directly. AMOS provides us with a powerful function called CHANMV for this purpose.

R=Chanmv(C)

CHANMV tests an object assigned to channel number C and returns one of two values depending on whether it is currently being moved the AMAL system. A value of True (-1)

indicates that the object is still moving, and a value of False (0) tells us that the movement commands have finally run out of steam. Let's assume we'd previously assigned our missile to channel number 1. We could now move this missile with a command such as:

```
Amal 1,"Move 0,-210,30"
Amal On 1
```

After this missile has been moved 210 units upwards, Chanmv would return a value of zero, and our missile could be safely reused. I'll now provide you with a new version of ALLOCATE which uses this instruction.

```
ALLOCATE2:
  ALL=0
  Rem Check though the missiles to find some which are unused
  For M=MFIRST To MLAST
  Rem Checks whether the missile is being moved by AMAL
  Rem I'm assuming that the Channel number is the same as the Bob
  Rem number. This is certainly true in my previous example
  If Chanmv(M)
        PMISS(ALL)=M
        Inc ALL
  Endif
  Exit If ALL=MN
  Next M
  Return
```

Just to round off this section, here are some additional missile systems for you to examine. These can be slotted directly into Example 3.3. Simply add in the ALLOCATE system, and replace the old MISSILE subroutine with the appropriate new routines. If you're feeling adventurous, you even can combine them to allow the player to fire off missiles all four directions at once! For example:

```
If Fire(1) Then Gosub MISSILE3:Gosub REAR_MISSILE: Gosub SIDE_SHOOT
```

Warning! Unless you make your attack waves pretty damn devastating, the player will now be able to annihilate the entire wave in a couple of seconds! Hah!

```
REAR_MISSILE:
  If Timer<TR+12 Then Return
  MN=1 : Gosub ALLOCATE:Rem Needs ONE missile
  If ALL=0 Then Return
  Rem RI holds the image number used for the REAR_SHOT
  Bob PMISS(0),X Bob(SHIP),Y Bob(SHIP)+20,RI
  Amal PMISS(0),"Move 0,180,60"
  Amal On PMISS(0)
  TR=Timer
Return
SIDE_SHOT:
  If Timer<TS+8 Then Return
  MN=2 : Gosub ALLOCATE:Rem Needs TWO Missiles
  If ALL<2 Then Return
  Rem SDI holds the image number used for the SIDE SHOT
  Bob PMISS(0),X Bob(SHIP)-20,Y Bob(SHIP)+10,SDI
  Amal PMISS(0),"Move -320,0,80"
  Bob PMISS(1),X Bob(SHIP)+20,Y Bob(SHIP)+10,SDI
  Amal PMISS(1),"Move 320,0,80"
  TS=Timer
  Amal On PMISS(0) : Amal On PMISS(1)
  Return
```

3.5.4 Enemy missiles

The enemy missiles can be generated using almost the same system. The only difference is that it's now up to our program to decide when a missile should be launched. As before, there are two basic approaches. The first is to assign a single missile to each individual alien in our attack wave. This missile can then be fired at random intervals by our program. The general technique can be seen from the following pseudo-code:

```
check firing rate
If alien fires
Then
      Choose one of the attacking ships
      If the current ship's missile is free for use
      Then
          Fire missile from ship
      Endif
Endif
```

This would translate into an AMOS Basic routine such as:

```
Rem Check firing rate
Rem Fire missile if RND(100) is less than value held in variable FRATE
If Rnd(100)<FRATE
  Rem Choose one of the attacking aliens
  Rem NALIENS is assumed to hold the number of aliens
  Rem AFIRST stores the Bob number assigned to the first Alien
  Rem BFIRST holds the Bob assigned to the first Aliens' missile
  AN=Rnd(NALIENS)
  Rem Get Bob number used by selected alien
  A=AN+AFIRST
  Rem get Bob number used by its ship
  B=AN+BFIRST
  Rem Find position of the alien on the screen
  XA=X Bob(A):YA=Y Bob(A)
  Rem Find the position of the aliens' missile
  YB=Y Bob(B)
  Rem only fire missile if the alien's on the screen
  Rem and the missile is not!
  If XA>=0 and YA>=0 and YB>=200
    Rem Fire missile downwards
    Bob B,XA,YA+20,A_MISS
    Rem I'm assuming that the AMAL Channel B has been previously
    Rem assigned to the missile
    Amal B,"Move 0,210,30"
    Amal On B
  Endif
Return
```

This system may be fairly crude, but it's relatively simple and very fast. Most of the instructions are concerned with choosing an alien and getting its associated missile number. The meat of the routine is in the last few lines, which checks whether the alien is visible on the screen and has an available missile. If so, it fires off the missile using an AMAL Move instruction. If we were running short of AMAL animation channels, we could also move the missile directly from AMOS Basic.

One obvious flaw with this technique is that the number of missiles on the screen depends entirely on the number of aliens. So the fewer aliens, the less missiles there are for the player to dodge! A more effective approach is to keep a pool of all the available missiles in an array.

This can be loaded at the start of the main loop using a simplified version of our original ALLOCATE routine.

```
Here's the relevant bit of pseudo-code:
Create a list of the free missiles
Repeat
Take each alien in turn
  If it's visible
    Then
     Check firing rate
     If it's time to fire
     Then
       Fire off a missile
     Endif
    Endif
  Subtract one from the number of missiles
Until there are no more missiles
```

Now for a couple of routines to handle this procedure.

```
ALLOCATE_ALIEN:
  Rem This is just a simpler version of the original ALLOCATE
  Rem it expects the missile routine to place the number of
  rem missiles required in MN
  Rem After the routine has done its work, ALL will be loaded with
  Rem the number of free missiles
  ALL=0
  Rem Check though the missiles to find some which are unused
  Rem BFIRST is the bob number used by the first alien missile
  rem BLAST is the number of the last one
  For M=BFIRST To BLAST
  BY=Y Bob(M):Rem Get the Y coordinate of the missile
  Rem Checks whether a missile has crossed the bottom of the screen
  If BY>200
     Rem load missile number into PMISS
        PMISS(ALL)=M
        Inc ALL
  Endif
  Exit If ALL=MN
  Next M
  Return

AFIRE:
Rem Fire off a salvo of alien missiles
For A=0 To NALIENS-1
Rem check firing rate
If Rnd(100)<FRATE and ALL>0
  Get coordinates of alien
  XA=X Bob(A+AFIRST):YA=Y Bob(A+AFIRST)
  Rem only fire missile if the alien's on the screen
  If XA>=0 and YA>=0
    Rem Fire missile using the bob number in PMISS
    Bob PMISS(ALL),XA,YA+20,AMISS
    Amal PMISS(ALL),"Move 0,200,30"
    Amal On PMISS(ALL)
    Rem Get the next missile in the list
    Dec ALL
  Endif
Endif
Next A
```

Each turn of the loop now gives all the current aliens an equal chance to fire off a missile. As the aliens are killed off, the number of missiles assigned to each alien increases dramatically, so the firing rate hots up. This pays real dividends towards the end of an attack wave, because the last alien is now every bit as challenging as the first.

3.5.5 Guided missiles

Guided missiles are also possible. In order to move one of these missiles across the screen, we need to get the relative distances between the missile and the intended target. These can be worked out using the X Bob and Y Bob functions like so.

The vertical distance between the missile and the target is just:

```
DY=Y Bob(Target)-Y Bob(Missile)
```

and the horizontal distance is:

```
DX=X Bob(Target)-X Bob(Missile)
```

From these figures, we can easily work out an interception course for our missile.

```
GUIDED:
 Rem GM is the Bob number assigned to the guide missiles
 XM=X Bob(GM) : YM=Y Bob(GM)
 Rem MT is the Bob number assigned to the target
 Rem DX and DY are to the movement direction needed
 Rem to move the missile closer to the target.
 Rem By multiplying these values by a fixed number
 Rem we can speed up the missile as required
 DX=Sgn(X Bob(MT)-XM) : DY=Sgn(Y Bob(MT)-YM)
 Bob 2,XM+DX,YM+DY,9
Return
```

This routine should be called once during every turn of our main program loop. The AMAL equivalent of this routine depends on whether the missile is aimed at an alien or the ship. It's either:

Attacking an alien

```
ZAP_ALIEN$="Loop; Move RX-X,RY-Y,16; Jump L"
```

where *RX* and *RY* are AMAL registers holding the coordinates of our selected target. As the coordinates change, their values need to be continually stuffed into *RX* and *RY*. We could be accomplish this by adding the following line to the end of our main program loop:

```
Amreg(23)=X Bob(TARGET):Amreg(24)=Y Bob(TARGET)
```

Attacking the player

```
ZAP_PLAYER$="Loop; Move XS(0,XM)-X,YS(0,YM)-Y,32; Jump L"
```

XM and *YM* are built-in AMAL functions which return the current hardware coordinates of the mouse, and *XS* and *YS* convert these coordinates into their normal screen versions. Note that this routine assumes that we've previously moved the mouse to the X and Y coordinates of our ship using the X MOUSE and Y MOUSE functions from AMOS Basic. So it's mainly designed to be used with my UCS system.

3.5.6 Lasers

Lasers are quite easy to create in AMOS Basic. One, admittedly crude, possibility is to assign a long, thin image to a Blitter object and flash it on the screen for a second or so. This produces a reasonable laser effect with practically no extra programming.

But if we want to create the illusion of an expanding beam, we'll need to do a bit more work. We simply take our previous Blitter object, and animate it through a series of larger and larger images using AMAL. This allows us to treat our laser as an extremely large missile, and enables us to use all our standard collision detection routines with impunity. Here's a small example routine to get you started.

```
LASER:
  Rem Check if laser is still growing
  If Chanan(LASE) or Joy(1)<>16 Then Return
  Rem Assumes first image starts at thirty
  Rem position laser beam at top of ship
  Bob LASE,X Bob(1),Y Bob(1)-16,30
  Rem Create an expanding laser effect with AMAL
  A$="Anim 1,(30,2)(31,3)(32,3)(33,3)(34,3)(36,3)(37,3); "
  A$=A$+"Loop : If A<37 Jump Loop ;Let X=-100;"
  Amal LASE,A$
  Amal On LASE
  Boom
Return
Procedure MAKE_LASER
  Cls 0
  Ink 11
  BL=30:Rem first image is 30
  Cls 0,0,0 To 32,32
  Rem create a series of images for a laser beam
  Rem Adjust number and size of images to taste
  For L=10 To 80 Step 10
  Ink 8 : Bar 6,0 To 10,L
  Get Bob BL,0,0 To 16,L
  Hot Spot BL,$12
  Inc BL
  Next L
End Proc
```

The example generated a vertical laser, but the same system could be easily modified to create a beam growing from left to right. You'd merely need to slightly change the values used by the Bar and Get Bob commands.

3.6 Moving the Aliens

The animation of the attacking aliens is another of the more vital areas of our program. If we make a mistake in this department, the game will be both ugly and unplayable. It's therefore well worth concentrating heavily on this section during the design of our game. This will pay off real dividends when we begin to generate the final AMOS Basic program. One quick word of warning though. Although hardware sprites work fine for the player's ship, their limitations normally make them totally unsuitable for the aliens. So unless you really enjoy a challenge, you are STRONGLY advised to use Blitter objects instead.

I'll now have a detailed look at several of the more useful techniques.

3.6.1 Moving the aliens with AMAL

The AMAL animation system provides us with a fast and simple way of moving our aliens across the screen. Well, three ways actually.

The easiest option is to exploit the same AMAL Move command we used for our missiles. Move may be only capable of moving our objects in a straight line, but with a little imagination it can also be used to generate some quite complicated attack formations. All we need to do is divide our pattern up into a series of straight line movements and assign each component to a separate AMAL Move instruction.

Let's demonstrate this technique with an example, showing how our aliens can be moved up and down the screen. The first step is to define our animation string. This will be split into two components like so:

```
Move 0,200,50
```

Moves the alien 200 units down in 50 steps

```
Move 0,-200,50
```

Move the alien 200 units up

If we want to repeat these movements, we'll need to enclose these commands in a loop. This results in a movement string such as:

```
S$="A: Move 0,200,50; Move 0,-200,50; Jump A"
```

The AMAL Jump command incidentally, is just the AMAL equivalent of the familiar AMOS GOTO instruction. It jumps straight to a label which we've defined previously in our AMAL program. The program now continues starting from the next instruction after the label. Anyway, we are now ready to set up the required AMAL animation channels and position our aliens on the screen.

```
Rem Assign Channels 8 through 15 to the aliens
Rem You can use any channels you like
For C=8 To 15
 Rem Assign Bob numbers 8 through 15 to aliens
 Channel C To Bob C


 Rem Now position each alien 30 units apart. This is essential,
 Rem otherwise all the aliens would be displayed on top of each
 Rem other!
 Bob C,(C-7)*30,0,AI
 Rem AI should be set to the image number used by your aliens


 Rem Set Up AMAL string
 Amal C,S$
Next C
Rem Start the animation
Amal On
```

Example 3.4: Moving the aliens with the AMAL Move command

```
Load "Extras:sprite_600/aliens/alien4.abk"
Double Buffer
Flash Off : Get Sprite Palette
Cls 0
AI=1
S$="A: Move 0,200,50; Move 0,-200,50; Jump A"
  For C=8 To 15
    Channel C To Bob C
    Bob C,(C-7)*30,0,AI
    Amal C,S$
  Next C
Amal on
Do
Loop
```

For our next example, we'll create a simple zig-zag effect. As before, we first need to define a couple of movement strings.

```
Move 600,0,100
```

Moves our aliens 600 units right in 100 steps.

```
Move -600,0,100
```

Moves our aliens 600 units left.

You may be wondering why I'm using such large values for the movement distances. After all, the screen is only 320 units wide. What's the point of moving our objects way past the edge of the screen? Well, there's a trick involved. In order to generate a nice smooth movement effect, we want to start our aliens off from the far left of the screen. You'll see how this works in a few moments.

So far, all we've done is move the alien from left to right. But I promised you a zig-zag, didn't I? This can be generated by enclosing the previous movement commands in a loop:

```
T: Move 600,0,100;    move each alien 600 units right in 100 steps
   Move -600,0,100;   move alien back to left hand corner of screen
   Let Y=Y+20;        go down twenty units
   Jump T;            repeat forever
```

The `Let Y=Y+20` command moves the alien 20 units downwards. It's similar to using `SY=SY+20:Bob 1,SX,SY,SHIP` in our Joystick routine.

Remember, AMAL automatically keeps track of our current object coordinates in two special AMAL variables called X and Y. We can assign these variables using the Let instruction to move our object around the screen. After the Move commands have done their work, the Let instruction moves the alien and the process repeats, moving the alien in a neat zig-zag pattern. Unfortunately, there's a snag. We've forgotten to check whether our alien has reached the bottom of the screen. If we run this program as it stands, the aliens will disappear completely after a couple of seconds. We can correct this by adding a simple test to our AMAL program.

```
T: Move 600,0,100;          move alien 600 units right in a hundred steps
   Move -600,0,100;         move the alien 600 units left
   Let Y=Y+20;              after each movement, go down twenty units
   If Y<200 then Jump T;    repeat until alien reaches bottom of the screen
   else Let Y=0; Jump T;    if so, restart alien back from top
```

Now for the finished program:

Example 3.5: A zig-zag pattern

```
rem Don't bother to type all this in.
rem You just need to change a couple of lines from the previous version.
Load "Extras:sprite_600/aliens/alien4.abk"
Double Buffer
Flash Off : Get Sprite Palette : Cls 0
AI=1
S$="T: Move 600,0,100; "
S$=S$+" Move -600,0,100 "
S$=S$+"Let Y=Y+20; If Y<200 Jump T; "
S$=S$+"Let Y=0; Jump T"
  For C=8 To 15
    Channel C To Bob C
    Rem position our aliens to the far right of the screen
    Bob C,(C-15)*30,0,AI
    Amal C,S$
  Next C
Amal On
Do
Loop
```

Can you see how I've initially assigned the aliens to negative coordinates? That's the trick I mentioned earlier. It starts the attack wave from the edge of the screen, and generates a delightfully smooth movement effect. The same technique can be used to generate hundreds of different attack waves.

Here's another AMAL routine, which moves the aliens through a diamond pattern:

```
S$="A: Move 150,-100,25; Move 150,100,25; Move -150,100,25; "
S$=S$+"Move -150,-100,25; Jump A "
```

This can be substituted directly for the AMAL command string in Example 3.5.

So much for the theory. Now's the chance for a bit of fun. Put down this book, load AMOS Basic, and have a bash at generating your own attack waves. Once you get the hang of it, it's surprisingly simple!

3.6.2 AMAL For..Next loops

It's also possible to animate our objects directly using the AMAL For..Next system. This allows us to create complicated movement patterns which change steadily over the course of time. The general idea is fairly obvious. It exploits the fact that AMAL keeps permanent track of the current position of our object in the variables *X* and *Y*. If we change these variables with the AMAL Let command, the object will immediately move to a new position. Here's an example:

```
Let X=160; Let Y=100;
```

This moves our alien straight to the centre of the screen. (coordinates 160,100)

As I showed you a few moments ago, we can also use the same technique to move the alien in a specific direction.

```
Let X=X+1;    Moves the alien one unit to the RIGHT
Let X=X-1;    Goes one unit to the left
Let Y=Y+1;    One unit Down
Let Y=Y-1;    Moves one unit Up
```

If we place a series of these commands in an AMAL For..Next loop, we can easily move our objects smoothly across the screen.

```
For R0=1 To 60      amal for next loop
Let X=X+4;          move current object four units right
Let Y=Y+3;          position object three units down
Next R0;
```

This generates a diagonal movement. The AMAL version of For..Next is more or less the same as its Basic equivalent. The only significant difference is that there's no STEP command. We are also limited to local AMAL variables from R0 to R9 or global variables from RA to RZ. Note that the *T* in *To* MUST be a capital letter, otherwise AMAL will generate an annoying error message. Ok?

We'll now enclose this example in a simple loop to create an attack wave.

```
Loop:
For R0=1 To 60;              repeat sixty times with R0 from 1 to 60
Let X=X+4;                   move each alien four units to the right
Let Y=Y+3;                   move alien three units down
Next R0;
Let X=X-240; Let Y=Y-180;    reposition alien when it reaches the bottom
Jump Loop;                   jump back to loop
```

After the For..Next loop has done its work, we can then reposition the alien at the top left of the screen and jump back to the start.

Example 3.6: Moving the aliens with an AMAL For..Next loop

```
Rem This is almost identical to the previous versions
Rem So you don't need to enter the whole thing!
Load "Extras:sprite_600/aliens/alien4.abk"
Double Buffer :Flash Off : Get Sprite Palette
Cls 0: AI=1
Rem New AMAL command string
S$="Loop: "
S$=S$+"For R0=1 To 60;
S$=S$+"Let X=X+4;"
S$=S$+"Let Y=Y+3;"
S$=S$+"Next R0;"
S$=S$+"Let X=X-240; Let Y=Y-180; Jump Loop"
For C=8 To 15
  Channel C To Bob C
  rem New starting position
  Rem moves the alien to a negative coordinate ranging from
  Rem -224 (7*-32) to 0
  Rem This starts the alien off the screen
  Bob C,(C-8)*-32,0,AI
  Rem start attack
  Amal C,S$
Next C
Amal On
Do
Loop
```

Now for a more practical example. We'll take a couple of simple For..Next loops and slowly expand them into a complex attack wave. Note: The following example programs are in AMAL rather than AMOS Basic. So in order to run them, you'll first need to copy them into the command string (*S$*) in Example 3.6.

Our starting point will be a pair of loops which move the aliens back and forth across the top of the screen.

```
Loop:                                   define an amal label
For R1=1 To 60; Let X=X+6; Next R1; moves 360 units right in 60 steps
For R1=1 To 60; Let X=X-6; Next R1; moves 360 units left
Jump Loop;                              jump back to loop and repeat
```

We'll can now extend this routine slightly to move the alien down a little after each loop:

```
Loop:
For R1=1 To 60; Let X=X+6; Next R1; move alien right
Let Y=Y+16;                 move the alien down sixteen units
For R1=1 To 60; Let X=X-6; Next R1; move alien left
Let Y=Y+16;                 move the alien down sixteen
Jump Loop;                  repeat forever
```

Currently, we haven't included any instructions to return the alien to the top of the screen after it disappears off the bottom. We can correct this omission by wrapping the movement commands in another For..Next loop like so:

```
Loop:
For R0=1 To 5;              move aliens back and forth 5 times
For R1=1 To 60; Let X=X+6; Next R1;sweep the aliens right
Let Y=Y+16;                 move the aliens down a bit
For R1=1 To 60; Let X=X-6; Next R1;sweep the aliens left
Let Y=Y+16;                 down again
Next R0;
Let Y=Y-160;                start the aliens back from the top
Jump Loop;                  repeat continually
```

Let's test this program from AMOS Basic and check out the results. Take the command string below, and type it straight into Example 3.6, replacing the original version of S$. When you run the program, the aliens will bounce across the screen in a simple attack formation.

```
S$="Loop: For R0=1 To 5;"
S$=S$+"For R1=1 To 60; Let X=X+6;"
S$=S$+"Next R1;"
S$=S$+"Let Y=Y+16;"
S$=S$+"For R1=1 To 60; Let X=X-6; "
S$=S$+"Next R1;"
S$=S$+"Let Y=Y+16;"
S$=S$+"Next R0;"
S$=S$+"Let Y=Y-160; Jump Loop;"
```

The next step, is to remove some of the *bounce* from this attack wave. It would certainly be nicer if the aliens could glide smoothly down the screen. After a little experimentation, you should discover that the bounce effect is actually caused by the lines containing `Let Y=Y+16`. These move the aliens 16 units straight down.

If we want to smooth off the edges of the pattern, we can replace these lines with the following Move commands.

Right hand edge

```
Move 16,8,4;          (Moves the alien diagonally)
Move -16,8,4
```

Left hand edge

```
Move -16,8,4
Move 16,8,4
```

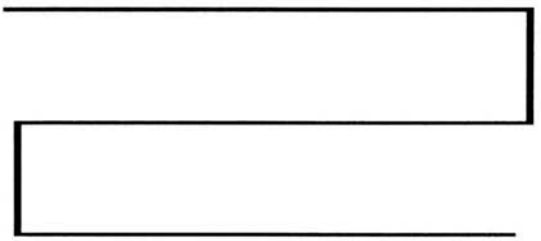

Figure 3.3: A simple attack wave

This results in a pattern as shown in Figure 3.4:

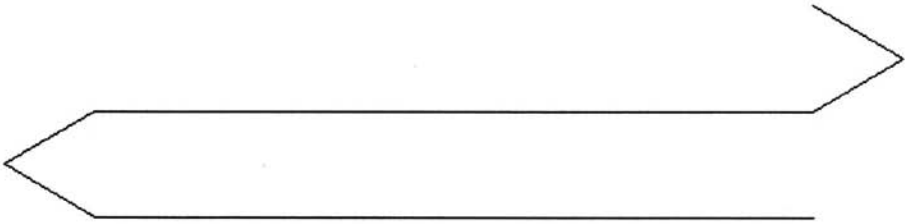

Figure 3.4: Smooth Zig Zag

Ok, so it's only a minor change. But the improvement in the attack pattern is very noticeable! Presently, all the aliens move together in a single mass. We can avoid this by launching the aliens onto the screen using a separate For..Next loop.

```
Loop: For R0=0 To R9*4; Let X=X+8; Next R0; increase separation by 8
For R0=1 To 5;                              main loop
For R1=1 To 60; Let X=X+6; Next R1;         sweep the aliens right
Move 16,8,4; Move -16,8,4;                  move the aliens down
For R1=1 To 60; Let X=X-6; Next R1;         sweep left again
Move -16,8,4; Move 16,8,4;                  down a bit
Next R0;
Let Y=Y-160;                                move alien to the top
Let X=R9*-32;    move alien to its original starting position
Jump Loop;                                  jump back to loop
```

Obviously, this program needs a little explanation:

R9 refers to an local AMAL variable and has a different value for each alien on the screen. It's loaded with the starting position of the current alien. Remember, this was initially displayed at the far left of the screen with:

```
Bob C,(C-8)*-32,0,AI
```

This places each alien 32 units apart from its immediate neighbours. So the first alien was displayed at (-224,0), the second at (-192,0) the third at (-160,0) etc.

The important part of this equation is the `C-8` bit, which is multiplied by -32 to get the starting coordinate of each alien. This value can be loaded into R9 with the AMREG instruction from AMOS Basic.

```
Amreg(C,9)=C-8:rem AMREG(channel,AMAL variable no)
```

After the attack wave's finished, our AMAL program can now neatly reposition each alien at its initial starting coordinates with the lines:

```
Let Y=Y-160;
Let X=R9*-32;
```

The next step, is to relaunch our aliens back onto the screen. Our AMAL For..Next loop just reads the value from R9, and uses it to slowly move the alien to the start of the screen. Let's replace the AMAL command string in Example 3.6 with our new code:

```
S$=S$+"Loop: For R0=0 To R9*4; Let X=X+8; Next R0;"
S$=S$+"For R0=1 To 5;"
S$=S$+"For R1=1 To 60; Let X=X+6;"
S$=S$+"Next R1;"
S$=S$+"Move 16,8,4; Move -16,8,4;"
S$=S$+"For R1=1 To 60; Let X=X-6; "
S$=S$+"Next R1;"
S$=S$+"Move -16,8,4; Move 16,8,4;"
S$=S$+"Next R0;"
S$=S$+"Let Y=Y-160; Let X=R9*32*-1; Jump Loop;
```

When you run this program, you'll quickly discover that the distance between the aliens seems to increase after every loop. This adds a much needed element of variety to the wave. It's also completely accidental!

The separation is actually generated automatically by the AMAL system! It's produced because of the relative slowness of our AMAL For..Next loops. By the time the first alien has been correctly positioned, it's an additional 24 units in front of the second. So whenever the aliens are relaunched, they move a further 24 units apart.

With a bit of experimentation, you can easily vary the distance between each alien in the wave. This can be changed by altering the values in the For..Next loop. For example:

```
For R0=0 To R9*4 ; Let X=X+8; Next R0;
```

produces a separation of 24 units

```
For R0=0 To R9*8 ; Let X=X+4; Next R0;
```

separates the aliens by 48 units

```
For R0=0 To R9*2; Let X=X+16; Next R0;
```

places the aliens 12 units apart.

Don't worry if all this sounds like gobbledegook. You don't need to understand any of it in order to exploit the idea in one of your games. You should however, be aware of the crucial importance of timing in an AMAL program. This can have a tremendous effect on the final results. It can also give you a severe headache when you're debugging your AMAL programs!

For a finale, we'll take the current program, and give it an extra little twist. Instead of moving each alien a fixed distance at a time, we'll change the pattern on the fly to create a neat sweeping effect as in Figure 3.5.

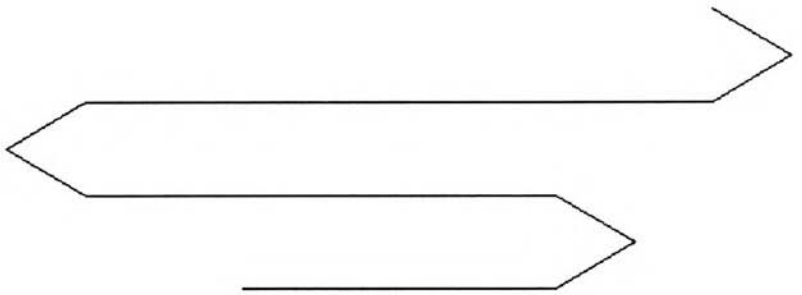

Figure 3.5: Sweeping the aliens across the screen

```
Loop: For R0=0 To R9*4; Let X=X+8; Next R0;  move alien into position
For R0=1 To 5;                               repeat five times
Let R2=R0*16;                                reduce size of movement counter
For R1=1 To 60-R2; Let X=X+6;Next R1;        move alien right by 60-R2 steps
Let R2=R2+8;                                 reduce size of movement counter
Move 16,8,4; Move -16,8,4;                   turn rightmost corner
For R1=1 To 60-R2; Let X=X-6; Next R1;        move alien left by 60-R2 steps
Move -16,8,4; Move 16,8,4;                    turn left corner
Next R0;                                      end wave
Let Y=Y-160; Let X=R9*32*-1;                  restart wave at top
Jump Loop;                                       repeat forever
```

The new program steadily increases the value of variable *R2* and uses it to slowly reduce the size of each individual movement stage. As the attack wave progresses, the R1 loop gets smaller and smaller, and the alien rapidly converges towards the centre of the screen. This effect is strongly reminiscent of the original Space Invaders.

Example 3.7: A complex AMAL attack formation

```
Rem It's quite similar to Example 3.6
Rem So you won't need to type everything
Load "Extras:sprite_600/aliens/alien4.abk"
Double Buffer
Flash Off : Get Sprite Palette
Cls 0
SHIP=1
rem Define the attack wave
S$="Loop: For R0=0 To R9*4; Let X=X+8; Next R0;"
S$=S$+"For R0=1 To 5;"
S$=S$+"Let R2=R0*16;"
S$=S$+"For R1=1 To 60-R2; Let X=X+6;"
S$=S$+"Next R1;"
S$=S$+"Let R2=R2+8; Move 16,8,4; Move -16,8,4;"
```

```
S$=S$+"For R1=1 To 60-R2; Let X=X-6; "
S$=S$+"Next R1;"
S$=S$+"Move -16,8,4; Move 16,8,4;"
S$=S$+"Next R0;"
S$=S$+"Let Y=Y-160; Let X=R9*32*-1; Jump Loop;"
Rem Set up the amal animation channels
For C=8 To 15
  Channel C To Bob C
  Bob C,(C-8)*-32,0,SHIP
  Rem start attack
  Amal C,S$
  Rem Load the alien number into R9
  Amreg(C,9)=C-8
Next C
Amal On
Do
Loop
```

Hopefully, you'll now have a good idea of the types of movements which can be generated by AMAL For..Next loops. The key thing to remember when you are creating these effects, is to experiment. You may go down the occasional blind alley, but when you are successful the effects can be terrific!

3.6.3 The Random element

One problem with all my previous attack waves, is that they are almost totally predictable. It would clearly be better if we could incorporate some sort of random element into our movement effects, so that the aliens moved differently each time our game was played.

As you may know, AMOS Basic includes a RND command which allows us to generate long sequences of random numbers. Is there any way we could use these numbers directly from AMAL? The obvious possibility would be to create our numbers with RND and then load them into an AMAL register using the Amreg function.

```
Amreg(0)=Rnd(100)
```

loads a random number from 0 to 100 into AMAL register RA.

Unfortunately, if we want to create truly unpredictable attack waves, we'll need to generate dozens of random numbers at a time. This would require us to load our values in some sort of loop, and would add an unacceptable delay to all our programs.

AMAL therefore provides us with a built-in random number generator called Zandom. This can be accessed directly inside our movement sequences using the Z(N) function. Zandom has been especially designed for maximum speed, so the numbers it produces aren't quite as random as the equivalent AMOS RND function. But as I'll be showing you in a few moments, it's more than adequate for most games.

In use, Z is just a little trickier than the equivalent AMOS Basic RND function. It actually just generates a number between -32767 and +32768. The value between the brackets, N, is then combined with this number using a logical AND operation to get the final random value.

If this sounds overly technical, then I've good news. Providing you choose the value of *N* carefully, it will work exactly like the familiar AMOS RND function. Here's a list of suitable numbers: 1,3,7,15,31,63,127,255. When we substitute these values for *N*, we get a series of numbers ranging from 0 to *N*. So:

`Z(3)` produces a random number from 0 to 3.

`Let X=Z(255)` moves our object to a random X coordinate from 0 to 255.

The reason for the odd name, incidentally, it that the developers had already chosen the letter R for the AMAL register variables. After several hours thinking about alternative names, they gave up in disgust and decided to call it Zandom instead. They then gleefully assigned it to the letter Z which was currently unused!

The Zandom command can be used directly in all AMAL commands, including Move, Anim or Let. So it's really surprisingly easy to generate completely random movement patterns. Example 3.8 contains a nice demonstration of these effects.

Example 3.8: Completely random movement

```
Rem Random Movements
Load "Extras:sprite_600/aliens/alien4.abk"
Double Buffer
Flash Off : Get Sprite Palette
Cls 0
ALIEN=1
Rem Random diagonal movements
Rem The pattern is similar to the movements of a gas molecule in an
Rem enclosed vessel. Respectable physicists call it a random
Rem molecular dispersion effect.
Rem Disreputable physicist's call it a Drunkards Walk, as it
Rem strongly resembles their own movements after they've had a
Rem few drinks!
Rem
Rem Start by getting the horizontal distance to be moved in R0
Rem values range from -64 to +64
Rem RX is loaded with the random X coordinate
A$="Horiz: Let R0=Z(15)*8-64; Let RX=X+R0;"
Rem Check whether the move will position the alien off the screen
Rem If so, jump back to Horiz and create another number
A$=A$+"If RX<0 then Jump Horiz;"
A$=A$+"If RX>320 then Jump Horiz;"
Rem Generate a random vertical distance in R1
A$=A$+"Vert: Let R1=Z(15)*8-64; Let RY=Y+R1;"
Rem Test new Y coordinate to see if it's acceptable
A$=A$+"If RY<0 then Jump Vert"
A$=A$+"If RY>200 then Jump Vert"
Rem Move alien to random screen position in sixteen steps
A$=A$+"Move R0,R1,16;"
A$=A$+"Jump Horiz;"
Rem Here's another one for the road
Rem This version moves our alien either vertically or horizontally
Rem by a random distance
Rem Choose a vertical or horizontal movement
B$="Loop: Let RF=Z(3); If RF<2 then Jump Horiz; else Jump Vert"
Rem Horizontal movement
Rem Get a random movement distance from +127 to -127
B$=B$+"Horiz: Let R0=Z(31)*8-127; Let RX=X+R0;"
Rem check for a legal move
B$=B$+"If RX<0 then Jump Horiz;"
B$=B$+"If RX>320 then Jump Horiz;"
Rem Move alien horizontally
B$=B$+"Move R0,0,16;"
Rem Handle vertical movement
B$=B$+"Vert: Let R1=Z(15)*8-64; Let RY=Y+R1;"
Rem Is move to a visible screen position
```

```
B$=B$+"If RY<0 then Jump Vert"
B$=B$+"If RY>200 then Jump Vert"
Rem move vertically
B$=B$+"Move 0,R1,16;"
Rem repeat
B$=B$+"Jump Loop;"
Rem set up AMAL animation sequence
For C=8 To 15
  Channel C To Bob C
  Rem position alien at random location
  Bob C,Rnd(320),Rnd(100),ALIEN
  Rem start attack
  Amal C,A$: Rem Use B$ for the second wave
Next C
Amal On
Rem start movement
Do
Loop
```

These AMAL routines choose a random distance in either the vertical or the horizontal directions, and check to see if the new screen coordinate lies inside the current screen. If not, they jump back to the Z() function and try again with another random number.

Note how I'm using SMALL values for *N* and multiplying them to get the final movement distance.

```
Let R1=Z(15)*8-64;
```

I could also have used `Let R1=Z(127)-64` instead, but this would have created distances which didn't divide evenly by the number of movement steps. So the resulting movement pattern would have lurched badly around the screen.

Try playing around with these equations on your own. You'll be able to generate dozens of attractive effects. One slight warning though. If you make a mistake in the formula, all your coordinate values might be rejected out of hand by the AMAL If commands. Occasionally, this will end up tying your AMAL program in an infinite loop. The result will in a massive slowdown in the AMOS system, and will even affect the positioning of your direct mode and editor windows when you leave the program. Don't panic however! Hit ESC and enter direct mode. After a few moments, the screen will slowly slide into place, and you can type `AMAL OFF` to restore the system to normal. Ah!

3.7 Movement Tables

AMAL attack waves are great fun, but believe me, AMOS Basic has also still got a few tricks up its sleeve. We can, for instance, also move our aliens directly from AMOS Basic using a movement table.

A movement table is just a simple list of coordinates, with one pair of numbers corresponding to each possible position of the alien. Every time the alien is moved, the appropriate entry in the movement table is read from an array to find the new screen position. It's that easy.

If we wanted to create one these tables in Basic, we'd first store all the coordinates in a couple of arrays such as:

```
Dim X(1000),Y(1000)
```

We could then move our aliens into place with a series of instructions like:

```
Bob 1,X(I),Y(I),1
```

This would be added INSIDE the main game loop. Here's an example of this system for you to type in:

Example 3.9: A movement table

```
Rem reserve some space for the arrays
Set Buffer 20
Rem defined arrays to hold the x and y coordinates of a point
Dim X(1000),Y(1000)
Rem create movement table
Gosub MTAB
Rem standard definition stuff
Double Buffer
Load "Extras:sprite_600/aliens/alien4.abk"
Flash Off
Get Sprite Palette
Curs Off : Cls 0
SHIP=1
Centre "<Movement Tables>"
Rem main loop
Do
   Rem move ship
   For P=1 To N
   Bob 1,X(P),Y(P),SHIP
   Rem add a (SMALL) delay between movements
   For W=1 To 30 : Next W
   Next P
Loop
Stop
Rem Create a simple movement table
MTAB:
N=320
For M=0 To N
X(M)=M : Y(M)=100
Next M
Return
```

The movement table is stored in the arrays X() and Y(). I've loaded these arrays with a series of coordinates which move the alien in a straight line. MTAB sets the X coordinates in X() from 0 to 320, and loads all the Y coordinates with 100. So the move ship routine can slowly sweep the alien from 0,100 to 320,100.

As it stands, it's all pretty boring, as we could have achieved the same effect with just a single AMAL Move command such as:

```
Move 320,0,320
```

We can however, use the same idea to generate fantastically smooth attack formations for our games. All it takes is a little mathematical wizardry, care of the Game Maker's manual!

3.7.1 Generating a movement table mathematically

If you've studied mathematics at school, you may remember two functions called Sine and Cosine. Up until now, you've probably thought these functions had something to do with triangles! Well, prepare yourself for a shock! We'll now hijack these functions to create some terrific attack waves. If we plot the values of Sine and Cosine on a graph, we get the curves in Figure 3.6.

Each point in the wave is described by a single pair of X,Y coordinates. The Y coordinate is produced by the Sine or Cosine function, and ranges from -1 to +1. The X coordinate is measured in degrees and can take any value we like.

Supposing we were to convert these numbers into their equivalent screen coordinates. We could now load these values into a movement table and flip our aliens through a smooth Sine wave. Rather than dwelling on abstract theory, I'll now give you a practical demonstration of the system in action. Take the program in Example 3.9 and replace the MTAB subroutine with:

```
MTAB:
Degree
N=0
D=360
For P=1 To D
  Inc N
  X(N)=N
  Y(N)=80*Sin(P)+100
Next P
Return
```

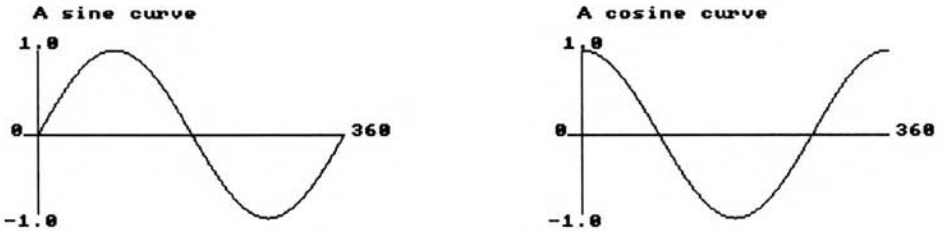

Figure 3.6: Sine and cosine waves

The key equation to look at is:

```
Y(N)=80*Sin(P)+100          Equation (1)
```

This takes the value generated by the Sine function, and converts it into a screen coordinate ranging from 20 to 180 as you can see in Figure 3.7.

P	Sin(P)	80*Sin(p)	80*Sin(p)+100
0	0	0	100
90	1	80	180
180	0	0	100
270	-1	-80	20
360	0	0	100

Figure 3.7: Conversion table

If we plot these values onto a graph, we get the results shown in Figure 3.8.

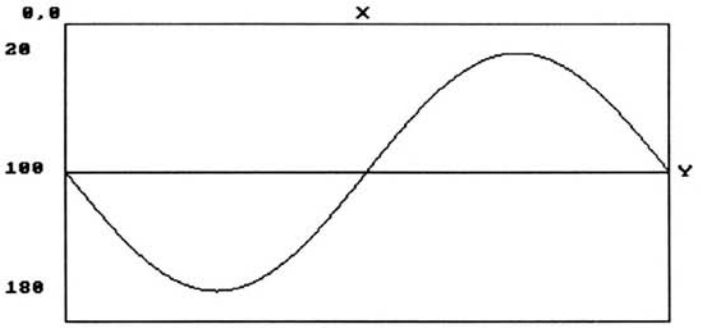

Figure 3.8: Moving the alien through a Sin wave

As you can see, the height of the curve is 80, and the centre has a Y coordinate of 100. This can be freely changed within our program to generate a whole family of different movement effects.

```
MTAB:
Degree
H=50 : Rem height of wave
CY=50 : Rem starting point of wave
N=0 : Rem counter
D=360 : Rem no of degrees
For P=1 To D
  Inc N
  X(N)=N
  Y(N)=H*Sin(P)+CY
Next P
Return
```

I've now replaced equation (1) with a more general version.

```
Y(N)=H*Sin(P)+CY        Equation (2)
```

H is the height of the wave and can be anything from 1 to 200.

CY holds the Y coordinate of the centre of the wave.

Try experimenting with different values of *H* and *CY* and you should discover a range of useful patterns. We can also flip the entire wave on its side. This can be achieved by just by swapping the X and Y coordinates around:

```
MTAB:
Degree
W=100 : Rem width of wave
CX=160 : Rem centre of wave
N=0 : Rem counter
D=360
For P=1 To D
  Inc N
  X(N)=W*Sin(P)+CX
  Y(N)=N mod 180
Next P
Return
```

W holds the width of the wave, and *CX* stores the X coordinate of the centre point. If you haven't done much programming, you may be wondering about the line:

```
Y(N)=N mod 180
```

Mod is a useful function which allows us to cycle a variable through a repetitive series of values. It works by returning the remainder after a division.

R=V mod D

is equivalent to:

R=V-Int(V/D)*D

Here are some examples:

0 mod 2 = 0-(0/2)*2 =0-0*2 =0
1 mod 2 = 1-Int(1/2)*2 =1-0*2 =1
2 mod 2 = 2-Int(2/2)*2 =2-1*2 =0
3 mod 2 = 3-Int(3/2)*2 =3-1*2 =1

As *V* steadily increases, *V mod D* therefore generates a series of numbers ranging from 0 to D-1. In our previous example, *N* increases by 1 after every loop. So *N Mod 180* repeatedly cycles from 0 to 179.

So far we've only used a single wave in our movement patterns. But as you can see from the diagram below, the sine wave actually repeats forever as the X coordinate gets bigger.

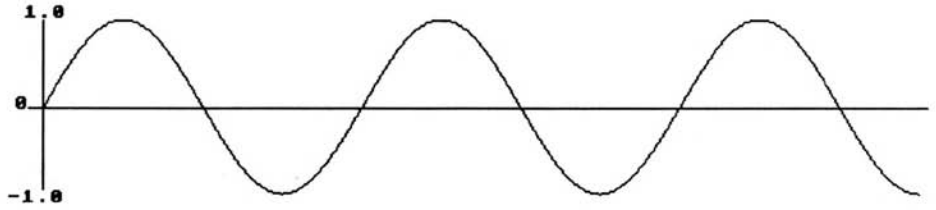

Figure 3.9: Fitting several waves on the screen

We can easily compress this pattern to fit several waves on the screen at once. Let's place the number of waves in a variable called *F*, for frequency.

Example 3.10: Pattern creator

```
Rem Don't bother to type all this in
Rem You only need to replace MTAB and the For W=1 To 40 loop
Rem from Example 3.9
Set Buffer 40
Dim X(1000),Y(1000)
Gosub MTAB
Double Buffer
Load "extras:sprite_600/aliens/alien4.abk"
Flash Off
Get Sprite Palette
Cls 0
Autoback 0
Bob Update Off
SHIP=1
Do
```

```
  For P=1 To N
   Bob 1,X(P),Y(P),SHIP
   Rem slow down the pattern as the frequency increases
   For W=1 To 40*F : Next W:Rem change this line from 3.8
  Next P
Loop
Stop
MTAB:
Degree
F=10 : Rem frequency of wave (waves per screen)
H=100 : Rem height of wave
CX=0 : Rem X coord of centre of wave
CY=100 : Rem Y coord of centre
N=0 : Rem counter holding the number of coordinates in the table
D=F*360 : Rem Get the length of the wave in degrees
Rem Sample one in F points from the wave and copy it into our table
For P=1 To D Step F
  Inc N
  X(N)=N+CX
  Y(N)=H*Sin(P)+CY
Next P
Return
```

F is the frequency of the wave. In our present example, it represents the maximum number of waves which will fit on the screen. Before continuing, have a play with the above example. Try changing the following variables:

H the height of each loop. Use values like 20,50,150,etc

CX,CY the screen coordinates of the wave's centre. The centre can be anywhere on the current screen. Try 50,50 or 160,100

F the number of waves per screen. Good values are 1, 2, 4, 8, 10 and 20!

We'll now extend these routines slightly to create circular or oval movement patterns. These can be generated from the following formulae:

X(N)=W*Sin(P)+CX *Equation (4)*
Y(N)=H*Cos(P)+CY *Equation (5)*

Don't worry if you don't understand how this works. It isn't necessary! But you need to know that:

W is the radius of the circle. That's exactly half its width.

H holds the height of the pattern divided by two. If it's the same as W then we'll get a circle, otherwise our objects will move in some sort of oval pattern.

CX,CY hold the screen coordinates of the centre of the pattern.

That's it! Figure 3.10 should make things a little clearer.

Replace the MTAB routine from Example 3.9 with:

```
MTAB:
Degree
H=50 : Rem set height of wave
W=100 : Rem set width of wave
CY=50 : Rem set centre of wave
CX=160 : Rem starting point of wave
N=0 : Rem counter
```

```
Rem if D this is less than 360, then the pattern moves through part of a
Rem circle.
D=360
For P=1 To D
  Inc N
  X(N)=W*Sin(P)+CX
  Y(N)=H*Cos(P)+CY
Next P
Return
```

Instant circles! As before, you should experiment with this routine carefully before continuing. There are literally hundreds of different possibilities to choose from. Try changing:

W,H Determine the size and shape of your movement pattern. Use values like (20,100), (100,20) or (100,100).

CX,CY Sets the coordinates of the centre of the pattern. Change CX to move the pattern right or left. Adjust CY to move the pattern up or down.

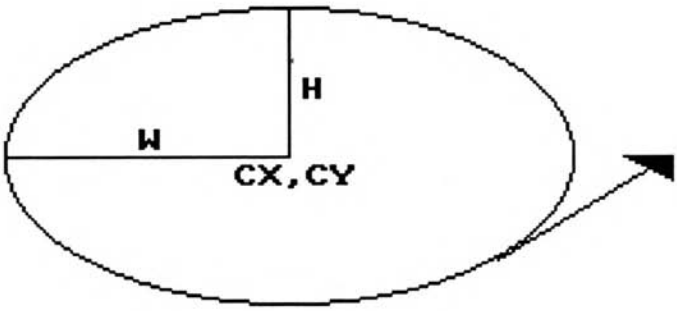

Figure 3.10: Circular movements

3.7.2 Using a movement table to move several objects

So far, we've concentrated on creating our movement patterns in memory. So we've restricted ourselves to moving a single object at a time. We can however, easily use the same trick to move an entire attack wave across the screen. The best way of achieving this, is to launch the aliens on the screen one after another. Instead of being released all at once, each alien waits a predetermined time before joining the pattern on the screen. This allows us to use the same movement table for all the aliens in an attack wave, saving masses of memory. All we need are a couple of arrays:

D Holds the delay between launching the selected alien on the screen, measured from the start of the wave. The units used to measure this delay vary from program to program, but the counter will usually be increased by 1 after each movement step. *NM* Stores the position of the alien in the movement table. This is simply the number of the next entry which will be read by the move routine. So the coordinates of alien B can be obtained using:

```
P=NM(B):BX=X(P):BY=Y(P)
```

The contents of these arrays could be defined using something like:

```
Rem set up attack wave
Rem NB=Number of Bobs to move
Dim D(NB),NM(NB)
For B=1 To NB
  Rem release aliens ten time units apart
  D(B)=B*10
  NM(B)=1 : Rem Set starting point in waveNext B
Return
```

We could now move our aliens with:

```
SHIP=1
SP=4:Rem Speed
Do
  For B=1 To NB
   Rem Wait for a launch
   If D(B)>0
     Add D(B),-1:Rem Subtract one from the counter until it's zero
   Else
     Rem Move alien
     P=NM(B):Rem Get position of next set of movement coordinates
     Bob B,X(P),Y(P),SHIP:Rem Move alien
     Add NM(B),SP,1 To N: Rem Cycle through movement table
   End If
  Next B
Loop
```

Most of this should be fairly self explanatory, but you may be wondering about the ADD statement towards the end.

```
Add NM(B),SP,1 To N
```

This takes the movement counter in *NB* and increases it by a value held in *SP*. *SP* holds the speed of the movement, and allows us to move our aliens through several steps at a time through the pattern. The higher the value, the faster the apparent movement. If you change *SP* during your program, you can speed up the attack wave as the game progresses. *N* stores the total number of coordinates in the movement table. So the `1 To N` bit cycles the aliens continuously through the pattern at the selected speed.

Here's a complete program which demonstrates this system:

Example 3.11: Moving lots of aliens with a movement table.

```
Set Buffer 40
Dim X(1000),Y(1000)
Dim D(16),NM(16)
Gosub MTAB
Double Buffer
Load "extras:sprite_600/aliens/alien4.abk"
Flash Off
Get Sprite Palette
Cls 0
Autoback 0
SHIP=1
SP=4
Do
  For B=1 To NB
   If D(B)>0
     Add D(B),-1
   Else
```

```
    P=NM(B) : Bob B,X(P),Y(P),SHIP : Add NM(B),SP,1 To N
   End If
  Next B
Loop
Stop
MTAB:
Degree
F=1 : Rem frequency of wave (in cycles per screen)
W=50 : Rem width of wave
H=100
CX=160 : Rem centre of wave
CY=100
N=0 : Rem counter
NB=5
D=F*360
For P=1 To D Step F
  Inc N
  X(N)=W*Sin(P)+CX
  Y(N)=H*Cos(P)+CY
Next P
Rem set up attack wave
For B=1 To NB
  Rem release 1/5 of a second apart
  D(B)=B*10
  NM(B)=1 : Rem get starting point in wave
Next B
Return
```

One side effect of the above system, is that it's easy to remove an alien from the movement list after a collision has been detected. We just set *NM* to zero, and change the start of the loop to:

```
  For B=1 To NB
   Rem Wait for a launch
   If W(B)>0 and NM(B)=0
     If NM(B)<>0: Add W(B),-1:Endif
   Else
     Rem Move alien
     P=NM(B):Rem Get position of next set of movement coordinates
     Bob B,X(P),Y(P),SHIP:Rem Move alien
     Add NM(B),SP,1 To N: Rem Cycle through movement table
   End If
  Next B
```

3.7.3 Smoothing the waves with UPDATE

If we're moving a lot of objects at once, we may find that there's a slight judder effect. As a default, AMOS automatically redraws each object on the screen whenever its coordinates are changed in our programs. This means that every time we move an object using the bob or Sprite commands, it will appear at the new position almost instantaneously. Normally, that's fine, but if we are moving our objects inside a FOR..NEXT loop, each object will be repositioned independently of its neighbours. So the first object will be completely redrawn before the next one has been moved.

Let's get back to the demo program I showed you in Chapter 1.

```
Load "Extras:sprite_600/aliens/alien1.abk"
Flash Off : Get Sprite Palette
Double Buffer
Cls 0
```

```
Do
Rem Move Blitter objects across the screen
For X=1 To 320
Bob 1,X,0,1
Bob 2,X,50,1
Bob 3,X,100,1
Next X
Loop
```

Each bob is redisplayed on the screen immediately, causing a noticeable jerk to the movement pattern. The solution is to turn off the automatic updating system, and redraw all the objects in a single go. This can be achieved with the help of the following commands:

UPDATE OFF Stops AMOS redrawing each object after a movement

UPDATE Redisplays all objects at their new positions. This updates both SPRITES and BOBS.

BOB UPDATE Same as UPDATE but only works for Blitter Objects

SPRITE UPDATE Displays the SPRITES which have moved.

In order to ensure maximum smoothness, it's essential to add a WAIT VBL command after redrawing the screen. This waits for the drawing operation to finish before continuing with the rest of our program. It also delays the program for approximately 1/50th of a second (1/60th for NTSC users). If required, we can compensate for this delay by increasing the speed of our movements slightly.

Here's some pseudo-code which illustrates how we would use this system in a real program.

```
Load screens, and activate double buffering
Update Off (Turn off bob and sprite updates)
: :
(Main program loop)
Do
 Repeat
   Move object to new position
 Until No more objects
Update:Wait Vbl (display ALL objects in one go)
Loop
: :
```

Example 3.12: Movement tables with BOB UPDATE

```
Set Buffer 40
Dim X(1000),Y(1000),W(16),NM(16)
Gosub MTAB
Double Buffer
Load "extras:sprite_600/aliens/alien4.abk"
Flash Off : Get Sprite Palette : Cls 0
Bob Update Off :Rem Deactivate automatic bob updates
SHIP=1 : SP=4
Do
  For B=1 To NB
   If W(B)>0
     Add W(B),-1
   Else
     P=NM(B) : Bob B,X(P),Y(P),SHIP : Add NM(B),SP,1 To N
   End If
  Next B
  Bob Update : Wait Vbl:Rem Redraw bobs at their new positions
Loop
```

```
Stop
MTAB:
Degree
W=50 : H=100: CX=160 : CY=100
F=1 : N=0 : NB=5 : D=F*360
For P=1 To D Step F
  Inc N : X(N)=W*Sin(P)+CX : Y(N)=H*Cos(P)+CY
Next P
For B=1 To NB
  W(B)=B*10: NM(B)=1
Next B
Return
```

3.7.4 The AMAL PLay command

Interestingly enough, Movement tables can also be used with AMAL. AMAL provides us with a special command called PLay which automatically moves an object through a predefined movement table stored in the AMAL memory bank number 4.

The AMAL movement table isn't just a simple list of coordinates.It's composed of a series of offsets. These are added to the current coordinates of each object to get its new position. Let's have a look at the equivalent AMOS Basic version. Supposing we wanted to move bob number 1. The code would be something like:

```
Bob 1,X Bob(1)+DX(P),Y Bob(1)+DY(P),SHIP
```

where *X Bob* and *Y Bob* return the bob's X and Y coordinates, and the *DX()* and *DY()* arrays hold the movement table. The AMAL PL command is much simpler. It's just:

```
PL n
```

where *n* is the number of a movement pattern stored in the AMAL memory bank.

Ok, but how do we create one of these patterns in the first place? That's easy. There's a powerful AMAL editor program on the AMOS Extras disk waiting for our use.

The AMAL editor is one of those programs you either love or hate. Although it's extremely well written, it's certainly not to everyone's taste. Personally, I prefer to enter my AMAL programs into command strings, and test them directly from AMOS Basic. This has the advantage of allowing me to use the familiar AMOS Editor, and avoids the need to learn a whole bunch of new commands. But it's also possible to enter our programs straight into the AMAL editor and save them in the AMOS memory bank. The one part of the AMAL editor which stands out from the crowd, is the PLay recorder. This allows us to sketch out an attack wave using the mouse, and save it for future use by the AMAL PLay command. I'll now run through the entire procedure.

- Load the AMAL Editor from the AMOS Extras disk, and run it from the Editor. Don't worry if the screen goes blank for a while. AMOS is simply testing the program for errors. If you've bought the AMOS compiler, you can compile the AMAL editor for extra speed. The improvement is astounding!
- The PLay Editor can now be entered either from the Edit menu or by simply pressing the F5 key from the keyboard. Along the top of the screen you'll see a list of digits. These set the number of the PATH we are going to define. Each number is read downwards. So the first number is 01, the second is 02, and so on up to 48. The current path can be selected by just clicking on a number with the left mouse button.

- Record the pattern by selecting the Record option from the Movement menu or pressing F1. Now carefully position the mouse pointer to the starting point of your movement, and enter the recording speed from the keyboard. This sets a delay in 50ths of a second (60ths for NTSC) between each offset value. So the higher the speed, the coarser the movement. The default is 1, which is fine for most purposes. Hit Return to start the recording.
- Now move the mouse through your movement pattern. When you've finished, press the Left mouse button to continue. You'll then be asked to enter the name of your pattern. This can be anything up to eight letters.
- You can now test your pattern with the Play back option from the movement menu. Click on the Left button when it's over. Once you're satisfied with the results, select the number of next pattern to be defined from the selection window at the top of the screen.
- Finally, press ESC to return to the main AMAL Editor screen and save your patterns to the disk using the SAVE BANK option from the Disk menu. Don't forget to hold down the right mouse key while you're using the menu, by the way, otherwise the screen will remain blank!
- Exit from the AMAL Editor with ESC.

Whew! That was FUN! (and that was SARCASM!) Let's have a bash at actually USING our path in a real program. As an example, I'll take the demo file from the AMOS Extras disk, and incorporate it into a new version of Example 3.11.

First, we'll need to define our AMAL command string.

```
G$="Loop: P; PL 1;"
```

The PL 1 command animates our object using path number one. This must have been defined previously with the AMAL editor and loaded into memory.

After the PL command has been initialised, the object starts moving immediately, and our AMAL program continues from the next instruction. This makes it easy to check whether the alien has reached the edges of the screen. We can now reposition our objects as required, and repeat the movements continuously.

The code is:

```
G$=G$+"If Y<200 then Jump Testx else Let Y=-32; Jump Loop"
G$=G$+"Testx: If X>0 then Jump Loop else Let X=X+340; Jump Loop"
```

Incidentally, the *P* stands for Pause and is equivalent to the AMOS Basic WAIT VBL command. It's used to limit the PL command to a maximum of one movement in every 1/50 of a second. This saves quite a bit of time for our Basic programs, and keeps the rest of our games running smoothly.

Example 3.13: Using the AMAL PL command

```
Rem This is very similar to example 3.10, so you won't need to type it
Rem all again!
Rem Define arrays
Rem X,Y = the initial position of each alien on the screen
Rem W = delay from the start of the program to the release of an alien
Rem L  set to 1 if alien is awaiting launch or 0 if it has been released
Dim X(64),Y(64),D(64),L(64)
SHIP=1
Double Buffer
Gosub MSET
```

```
Load "extras:sprite_600/aliens/alien4.abk"
Flash Off : Get Sprite Palette : Cls 0
Rem Main program loop
Do
  For B=1 To NB
   Rem wait for a launch
   If D(B)>0 Then Add D(B),-1
   Rem Launch alien
   If D(B)=0 and L(B)=1
     Bob B,X(B),Y(B),SHIP : L(B)=0
     Amal On B : Rem Activate AMAL program assigned to alien
   End If
  Rem See section 3.7.5 for an explanation
  Rem Synchro:Wait Vbl
  Next B
Loop
Stop
MSET:
NB=5 : Rem Set number of bobs
Rem This command is used in the next section
Rem Synchro off
Rem Define Amal command string
G$="Loop: P; PL 1;":Rem Play path
Rem Test if the alien has passed the edge of the screen
Rem Reposition if necessary
G$=G$+"If Y<200 then Jump Testx else Let Y=-32; Jump Loop"
G$=G$+"Testx: If X>0 then Jump Loop else Let X=X+340; Jump Loop"
Rem Load the AMAL example file from the AMOS extras disk
Load "Extras:Amal_Example_File.abk"
For C=1 To NB
  Rem Assign an AMAL Channel to each object
  Channel C To Bob C
  Rem Position the object off screen
  Bob C,-100,-100,SHIP
  Rem Define AMAL movement.
  Amal C,G$
  Rem Set initial position of alien
  X(C)=C*16+150 : Y(C)=C*4
  Rem release about 1/5 of a second apart
  D(C)=C
  Rem Set Launched variable to one
  L(C)=1
Next C
Return
```

Hopefully, much of this example should seem pretty obvious. The X,Y arrays hold the starting coordinates of each alien on the screen. This is necessary because all movements are relative to the current object coordinates. If you change these values, you can generate dozens of useful effects. For example:

```
X(C)=C*4+150 : Y(C)=C*2 : Rem Creates a WORM effect
```

Also note that the AMAL_Example file only demonstrates a fraction of PLay's capabilities. So feel free to use your own patterns instead. With a little imagination, you should be able to produce some lovely attack waves with this system.

3.7.5 Beating the 16 object limit with AMAL

I'll conclude this section with a few words about the AMAL. Generally, AMAL only allows us to animate a maximum of 16 objects at a time. If we are moving a lot of objects, we may well run out of animation channels.

One possibility is to replace some of your AMAL movement routines with movement tables. As I showed you in Section 3.5.2, this is particularly easy with missiles, which only need to move in simple straight lines.

Alternatively, we can extend the AMAL system using the powerful SYNCHRO command. This turns off the normal interrupt routines and allows you to call AMAL directly in your Basic program. We can now control up to 64 objects at once!

SYNCHRO OFF

Switches off the AMAL interrupt system and lets you define channel numbers from 0 to 63!

SYNCHRO

This is used inside your basic programs, and executes a couple of instructions from every current AMAL program.

SYNCHRO ON

Restores the system to normal. Like UPDATE, SYNCHRO should usually be followed by a WAIT VBL. This synchronises the movements with the screen, and ensures the maximum smoothness,

Now for a QUICK example. Remove the Rem statements from the following lines in Example 3.10:

```
Synchro Off
Synchro: Wait Vbl
```

Now change the number of bobs to 63 with:

```
NB=63:Rem it's just after MSET:
```

and set the starting position of the aliens with:

```
X(C)=C*4+150 : Y(C)=C*2 : Rem Creates a WORM effect
```

(Well, I said it was quick, didn't I?)

When you run the updated program, you'll see a strange worm crawling across the screen. It's actually composed of 63 separate objects, moving together in formation!

Note that the decision to use SYNCHRO in your own games is NOT a foregone conclusion by any means. This is especially true if you're subsequently intending to compile your programs. When movement tables are compiled, they are often faster than the equivalent AMAL routines. So although AMAL is extremely useful, it's dangerous to go overboard and use it for everything. In my experience, the best results are obtained from a combination of both systems.

3.8 Animation Effects

So far I've been assuming that we will be using a single image for each alien in our attack wave. This is seriously boring! With a little extra programming, we can effortlessly animate our objects through a sequence of images.

3.8.1 AMAL Animation

In Chapter 1, I briefly introduced you to a powerful Animation command called Anim. The format of this instruction is:

```
Anim count,(image1,time1)(image2,time2)(... ...)
```

count is the number of times the entire sequence should be repeated. A value of zero repeats the animation continuously. This could be used for the animation effects for our aliens.

image1 is the number of an image from the sprite bank which is to be flashed on the screen.

time1 holds the time the image will be displayed, measured in 50ths of a second.

The really clever thing about Anim is that it can freely be combined with any of our AMAL movement patterns. We simply place the Anim at the start of the AMAL command string and move our objects as normal. The animation will now begin, and our AMAL program will immediately jump to the next AMAL instruction. For example:

```
S$="Anim 0,(1,1)(2,2)(3,3);"
S$=S$+"A: Move 150,-100,25; Move 150,100,25; Move -150,100,25; "
S$=S$+"Move -150,-100,25; Jump A "
```

This system makes it very easy to add real animation to our attack waves. A further discussion of the AMAL animation system can be found in Chapter 8 (Animation Effects).

3.9 Collision Detection

Once we've animated our aliens, moved our player's ship, and fired off the missiles, we'll obviously need to be able to detect the inevitable collisions. AMOS provides us with several dedicated commands. These are executed extremely quickly, allowing us to detect most collisions almost instantaneously.

Usually, we'll need to use one of the following three commands.

=Bob Col

Detects for a collision between two or more Blitter objects

=SpriteBob col

Checks for a collision between a sprite and several bobs

=Col

Holds the status of each bob after a collision.

That's about it! I've included the Sprite Bob command for those of you who wish to use a sprite for the player's ship. I'll now present you with a couple of cook-book solutions which will allow you handle most types of collisions reasonably efficiently. These can be freely modified for use with your own programs.

3.9.1 Detecting collisions with the player's ship

This is easy, as we can do the whole thing with a single BOB COL command.

C=Bob Col(S)

Checks the player's ship (bob number *S*) for a collision between any other current Blitter object. If a collision has been detected, Bob Col returns a value of TRUE (-1). We can test for this by using a simple IF..THEN statement in the form:

```
If Bob Col(S) Then Gosub DEATH
```

where *DEATH* is a routine to handle the explosion. A typical definition would be:

```
DEATH:
Boom:Rem Explosion effect
LIVES=LIVES-1:Rem Lose a life
Channel 1 to Bob S:Rem set up explosion sequence
Amal 1,"Anim 1,(9,1)(11,1)(13,1)(15,1)(17,1)(19,1)"
Amal On 1
Wait 20:Rem Wait for explosion to finish
Bob S,-100,-100,: Rem Move Ship off edge of screen.
Return
```

In many circumstances that would be all we need. But if we've allocated extra fire-power to the player, we might want to make sure that he hasn't collided with one of his own missiles. That would be too cruel!

Fortunately, we can restrict the test to a specific group of objects by extending the command like so:

```
C=Bob Col(S,FIRST to LAST)
```

where *FIRST* is the bob number of the first object to be checked for a collision with *S*. *LAST* is the bob number of the last object.

We'd need to number our aliens carefully to take advantage of this system. Suppose we had allocated bob numbers 10 to 19 for our aliens and 20 through 29 for our enemy missiles. We could now perform the entire collision check with:

```
If Bob Col(S,10 To 29) Then Gosub DEATH
```

Note: If you're using a sprite for your ship instead of a bob, you can substitute the SPRITEBOB COL function instead. Warning! You'll also need to add a MAKE MASK command at the start of your program, otherwise, the collisions just won't be detected.

3.9.2 Collision between attacker and a missile

Since this is a little complicated, we'll start off by examining the case of a single missile. We'll then extend our routine to handle multiple missiles. As before, the collision could be detected using a simple BOB COL command. S would be replaced with the bob number assigned to the missile, and FIRST to LAST with the numbers of the aliens. Taking our previous example, this would result in the following test:

```
If Bob Col (M,10 to 19) Then Gosub EXPLODE
```

M would hold the bob number assigned to our current missile. Note the difference in the range. I've intentionally ignored collisions between the player's missile and the alien missiles, as they're not important.

We now need to find out which alien had collided with our missile so we can destroy it. Boom! This can be found using the COL function which returns the status of each bob in our detection list after a BOB COL command. The format is;

C=COL(A)

where *A* is the number of the bob we wish to check. *C* is either TRUE (-1) or FALSE (0) depending on whether a collision has been detected between the missile and the alien.

We could now check each alien to see whether it had collided with our missile using a FOR..NEXT loop.

```
DEAD=0
For A=10 To 19
  If Col(A) Then DEAD=A
Next A
If DEAD Then Gosub DESTROY
```

where *DESTROY* would be a predefined AMOS Basic subroutine handling the explosion and increasing the score. Here's a typical definition for you to look at.

```
DESTROY
Boom:Rem Explosion effect
SCORE=SCORE+100:Rem Increase score
Channel 1 to Bob DEAD:Rem set up explosion sequence
Amal 1,"For R0=9 To 15; Let A=R0; P; Next R0; Let X=-1000;"
Amal On 1
Return
```

This uses an AMAL For..Next loop to generate an explosion effect and the moves the alien off screen with the `Let X=-1000` instruction. You could easily replace this with anything else you like.

The above system is totally compatible with the multiple missile routine I showed you earlier.

If you were using movement tables, you'd also need to load current position in *NM* with a value of zero as follows:

```
NM(DEAD)=0
```

This would stop the alien being moved after it's been destroyed. Similarly, if you're using AMAL, you can keep track of each aliens status in a separate array:

```
Dim ACTIVE(10)
```

Each alien would have its own individual entry in this array, As the level starts, all the values would be loaded with 1. Whenever an alien was destroyed, the approrpriate value could be set to zero. We could now restrict our collision tests to the moving aliens, saving oodles of time.

Remember all this only applies to a single missile. If we want to check several missiles, we'll need to use a series of BOB COL commands like so:

```
CDETECT:
Rem MFIRST=bob no of first missile
Rem MLAST=bob no of last
For M=MFIRST To MLAST
Rem AFIRST = number of the first alien to be checked
Rem ALAST = number of the last alien
   If Bob Col (M,AFIRST to ALAST)
      For A=AFIRST To ALAST
        If Col(A) and ACTIVE(A) Then DEAD=A: Gosub DESTROY
      Next A
   Endif
Next M
Return
```

If we're moving a lot of objects on the screen at once, the number of collision checks can easily get out of hand. Although each test is performed extremely quickly, there's a definite limit to the number of tests we can perform without causing annoying delays in the execution of our program. If you've access to the AMOS Basic compiler, you can obviously improve things quite a lot. But it still makes sense to reduce the collision system down to an absolute minimum. The faster we can make the collision routines, the better.

Number of tests=(Number of Missiles)*(Number of Aliens)

Let's assume we had 10 missiles and 10 aliens. The total number of checks would now be 10*10 or 100. That represents a pretty hefty burden on our program.

In practice, there are several strategies we can use to improve the efficiency of these routines.

One possibility is to ignore the missiles and concentrate on the aliens instead. If we are prepared to allow our missiles to continue after a collision, we can limit our checks to the various aliens. This results in a piece of code such as:

```
For A=AFIRST To ALAST
 If Bob Col(A,MFIRST to MLAST) Then DEAD=A: Gosub DESTROY
Next A
```

Despite the limitations of this system, it certainly has its uses. What's more, it's approximately 10 times faster than the previous version! It's especially good at handling the gun emplacements along the sides of a screen scroller. If we plan our missile movements carefully, we can often arrange it so that the missile disappears naturally by the time it hits a wall.

If we're feeling generous, we can tell the players that the missile has been absorbed into the rock. We can now position our aliens just above the absorption point, and test for the collisions at maximum speed. The rest of our aliens can be checked using the original detection routine.

This approach illustrates that we don't need to handle all collisions using a single routine. Providing we can ensure that a certain series of missiles can only hit some of the aliens, we can make massive time savings. Gun emplacements along the sides of the screen can only by hit by side shots. And if our aliens never move behind the player's ship, we don't need to check them for collision with our rear guns.

Let's assume we split our 10 aliens and 10 missiles into two groups of five. Each group would have its own separate collision detector. So the total number of checks would be:

(No of aliens in group 1)*(No of missiles in group 1)
+
(No of aliens in group 2)*(No of missiles in group 2)

or 5*5 + 5*5 = 25+25 = 50

Hey presto! We've just doubled the speed of our collision detector. Oddly enough, we could handle this with the same DETECT routine I showed you earlier. All we'd have to do would be to change the values of *MFIRST, MLAST, AFIRST* and *ALAST* inside our program. We could now call the same routine twice, to handle all our collisions.

Here's a fragment of code which demonstrates how it would work out.

```
MFIRST=10:MLAST=14:AFIRST=20:ALAST=24:Gosub CDETECT:Rem Group 1
MFIRST=15:MLAST=19:AFIRST=25:ALAST=29:Gosub CDETECT:Rem Group 2
```

Note that this idea will ONLY work if we could keep the missiles in each group completely separate. So we'd also have to change *MFIRST* and *MLAST* before using the ALLOCATE routine I showed you in Section 3.5.3.

3.10 Advanced Arcadia

3.10.1 Collision detection with AMAL

If you've examined the example programs included with AMOS Basic, you'll notice that they tend use AMAL for all their collision detection. In my experience though, the AMAL routines are often far more trouble than they are worth. There's nothing wrong with their implementation of course. After all, they WERE written by Francois Lionette! The problems only arise when we try to use them in an actual game.

Although AMAL is very powerful, it has no facilities for generating sound effects or incrementing high score tables. This forces our AMAL programs to communicate with AMOS Basic after a collision has been detected, and causes endless difficulties. Anyway, just for completeness, I'll now quickly run through the AMAL collision detection system.

Before starting, we need to turn off the automatic object movements with SYNCHRO OFF. The AMAL collision routines uses the Blitter and cannot be performed using the standard interrupt system.

We can now test for a collision using the AMAL BC function like so:

```
=BC(b,f,l)
```

Tests for a collision between two or more bobs where *b* is the number of the bob we wish to check. This bob is checked for collisions between bobs *f* to *l*. As you can see, BC is very similar to the AMOS Basic Bob Col command I showed you earlier.

We can call this command from within our AMAL program using a line like:

```
A$=A$+"If BC(1,10,19) then Jump Explode"
```

Note that I'm assuming that the missiles use bob numbers from 10 to 19. Naturally, you should substitute your own numbers for these values instead.

Explode is a LABEL inside our AMAL program. It might be defined as:

```
A$=A$+"Explode: Anim 1,(9,1)(11,1)(13,1)(15,1)(17,1)(19,1); P P P P P"
```

where the P commands force AMAL to wait until the animation has finished.

That would be fine for the ship. But what about the aliens? In this case, we'd need to keep track of the bob number assigned to each alien in our movement pattern so we can test it for collisions. One option might be to generate separate animation strings for each alien, and then insert the bob number during our program's initialisation phase.

We could also load the number of the current bob directly into an AMAL register. Remember that AMAL uses TWO types of variables.

R0-R9 are local registers and have a different set of values for each individual animation channel.

RA-RZ are global registers, and retain the same values between channels. So one set of registers suffices for the entire system.

If we want to save the bob used for the current alien, we'll need to place it into a local register such as R9. This should be done when we are defining our animation strings. For example:

```
For C=20 to 29
 Rem Since we're using synchro, we can use the same number for channels
 Rem and the bobs
 Channel C to Bob C
 Amal C,A$
 Amreg(C,9)=C
Next C
```

We can now replace the previous collision detector with:

```
A$=A$+"If BC(R9,10,19) then Jump Explode"
```

As it stands, the routine would be pretty useless, as we've no way of generating the sound effects and increasing the score. We therefore need to inform our AMOS Basic program that a collision had been detected. This could be achieved by simply loading a value into one of the global variables. Here's the new code:

```
A$=A$+"Explode: Let RC=1;Anim 1,(9,1)(11,1)(13,1)(15,1)(17,1)(19,1); P P P P P"
```

We could now detect the collision from AMOS Basic using the AMREG function.

```
Rem Generate BOOM,increments score and resets RC to zero
If AMREG(2)=1 Then Boom:Add SCORE,10:AMREG(2)=0
```

(AMREG(2) reads Register RC. RA=0,RB=1,RC=2 and so on until RZ=25)

The next step is to turn off the AMAL movement routine for our alien after the explosion. This saves valuable time for the rest of our program. Since we've already placed the bob number in register R9, we can load this into a global register such as RN, and access it from AMOS Basic.

```
A$=A$+"Explode: Let RC=1; Anim 1,(9,1)(11,1)(13,1)(15,1)(17,1)(19,1); P P P"
A$=A$+"P P P ; Let RN=R9"
```

We can now incorporate this procedure into our AMOS collision system like so:

```
If AMREG(2)=1
    Boom
    Add SCORE,10
    AMREG(2)=0
    Amal Off Amreg(13):Rem RN is the thirteenth global register
    Bob Off Amreg(13)
Endif
```

If you think that's complicated, I've bad news. There's more! We still haven't found out which missile has collided with the alien. Each missile will need its own separate BC command inside its movement string. The best way of accomplishing these tests is to move our missiles using AMAL For..Next loops rather than the more efficient Move command.

For example:

```
M$="For R0=0 To 200; Let Y=Y-2; If BC(R9,20,29) then Jump Bye"; Next R0;"
M$=M$+"Bye: Let Y=-200;":Rem Move missile off the screen
```

The numbers 20 and 29 refer to the first and last bobs used by our aliens. You could obviously change these to any values you wish

We could also achieve the same effect by assigning a separate animation channel for each alien just to handle the collisions. This would allow us to move our aliens using any AMAL commands we liked. But it's not recommended as it's pretty sluggish.

After all that hassle, you'd naturally expect a massive pay-off. Unfortunately, there often simply isn't one. Once AMAL has performed the collision detection, there's very little time for the actual movement effects to take place. So the entire game slows down significantly, even when it's compiled. It's even worse if we are managing a lot of objects. In this case the rest of our program can be reduced to a crawl.

Of course, I'm being intentionally pessimistic. If we are prepared to work at it, we can actually get some exceptional results from this system. If for instance, we're already using AMAL to control the player's ship, we can usually add collision detection in just a couple of lines. It's only when we try to handle the aliens that the complications set in.

A good demonstration of AMAL collision detection at its best, can be found in the Amosteroids program, or the Shoot-'em-up demo on the AMOS Extras disk. The programming may be convoluted, but it does work. So if you enjoy a challenge, feel free to experiment with this system to your heart's content. Otherwise, restrict your use of AMAL to movement effects and take things easy!

3.10.2 Animating the background

In many games, it's necessary to set our action against a rapidly changing background. A screen scrolling game, for instance, will be continually moving the landscape underneath the player's ship. Although I'll be discussing the details of these routines in Chapter 7, it's useful to have a general idea of the basic principles. This will pay off handsomely, when you get down to adding a little animation to your current playing area.

I'll commence with a bit of theory. Have you every wondered how the AMOS Blitter objects are actually moved? They seem to glide across the screen by magic, ignoring any existing graphics. But how does this really work?

Here's a quick explanation. The first thing to understand, is that Blitter objects are stored as part of the current screen. Bobs therefore completely overwrite anything underneath them on the display. A copy of the original screen area is kept automatically in the Amiga's memory. So when the bob is moved, the old screen data can be redrawn at its old position.

Of course, the actual sequence is rather more complicated due to the double buffering system used by AMOS Basic, but the principle remains the same.

Whenever a bob is moved, the following process takes place automatically.

- The area underneath the Bob is completely redrawn from the copy held in memory.
- A copy is then made of the section of the screen at the bob's new position.
- The bob is now redrawn at the new position, totally destroying the existing image.

Normally, all this is completely irrelevant, as the whole process is automatic. The snag occurs if we want to redraw the background screen behind our Blitter objects. In this case, we need some way of synchronising the bobs with our screen routines. AMOS Basic includes a couple of simple functions which allows us to control the entire updating process directly.

BOB CLEAR

Removes all Blitter objects from the screen and restores the display to its original, pristine condition. It's equivalent to Step 1. We can now change the screen to anything we like, without interference. Once we've finished, we can redraw all the bobs at their new positions with BOB DRAW.

BOB DRAW

Performs steps 2 to 3, and loads a copy of the screen area underneath the bobs, before redrawing all them on the display.

Together BOB CLEAR and BOB DRAW are equivalent to a single BOB UPDATE command. As with BOB UPDATE, the standard updating system needs to be deactivated with UPDATE OFF.

Here's a little pseudo-code which demonstrates this procedure.

Turn off Bob Updates with UPDATE OFF
Main program loop
Remove all the Blitter objects from the screen with Bob Clear
: :
Redraw screen
Move our objects
etc
: :
Replace our objects at their new positions with a Call to Bob Draw
Screen Swap:Wait Vbl
: :

The SCREEN SWAP command is important, as it displays the invisible logical screen under construction and hides away the old physical screen in memory. Don't worry if these terms are unfamiliar. I'll be explaining them carefully when I discuss Screen Scrolling in Chapter 7. For the present, just remember to include a SCREEN SWAP after a BOB DRAW, and everything will be hunky-dory.

Example:

```
Load "df1:Sprite_600/aliens/alien4.abk"
Load "df1:Sprite_600/space/ship1.abk",1
Double Buffer
Curs Off : Flash Off
Get Sprite Palette
Cls 0
Ink 31,0
Paper 0 : Locate 0,0 : Centre "<Background Animation> "
IMAGE=20
Hot Spot IMAGE,$11
Hide
DX=2 : DY=2 : SH=32 : SW=32
TX=0 : TY=128 : BX=320 : BY=192
SX=TX : SY=TY
TX=TX+SW/2 : TY=TY+SH/2 : BX=BX-SW/2 : BY=BY-SH/2
Limit Mouse X Hard(TX),Y Hard(TY) To X Hard(BX),Y Hard(BY)
P=1
Update Off
Autoback 0
Do
  Rem Remove SHIP from the screen
  Bob Clear
  Rem Animate screen
  Rem There are NO aliens, and no Blitter objects in this pattern
  Rem The only Bob on the screen is the SHIP!
  Rem I'm using the images as FILL pattern and then drawing a large
  Rem filled box. Ok so the effect is TOTALLY useless! But it's
  Rem certainly fun!
```

```
  Add P,1,1 To 4
  Rem Try this for an animating explosion
  Rem Add P,1,9 To 19
  Rem Set the fill pattern to image P from the sprite bank
  Set Pattern -P
  Rem Draw a box full of aliens!
  Bar TX,TY To BX-1,BY-1
  Rem read the mouse!
  Gosub MCS
  Rem Redraw the ship at its new position
  Bob Draw
  Rem Switch screens
  Screen Swap : Wait Vbl
Loop
Stop
MCS:
  SX=X Screen(X Mouse) : SY=Y Screen(Y Mouse)
  Bob 1,SX,SY,IMAGE
Return
```

Hint's and tips

- For the best results, get hold of the AMOS Basic compiler. The speed-up can be dramatic!
- If you're moving your objects with AMAL, turn off all the animation channels with AMAL OFF before starting each level. Only activate the channels which are actually being used by the current level.
- AMAL may be powerful, but there's a price to pay: the more AMAL channels you use, the less time will be available for your Basic programs. Sometimes, this can have a real impact on your games. If you're moving your missiles with the AMAL Move command, try moving them directly from AMOS Basic. You may notice a marked speed improvement!
- If you want to keep your ships and aliens in separate files, you can load them in two phases:

```
Load "aliens.abk"
Load "ships.abk",1
```

The , 1 bit just tells AMOS to add the new images to the end of the existing ones.

- Adding a hi-score table. This is complicated by the fact that you need to sort all the scores in descending order. The solution is to combine the names and the scores into a single HI_SCORE$ array, and sort them using the AMOS Sort commands. Here's a small demo to get you started!

```
Dim SCORE(11),NAME$(11),HI_SCORE$(11)
Rem First create some random names and random scores
For C=1 To 10
SCORE(C)=Rnd(10000) : NAME$(C)="Test_name"+Str$(C)-" "
Next C
Rem Now load these values into a single HI_SCORE array
N=6 : Rem This is the maximum number of digits in the score
For H=1 To 10
  Rem pad out the score with zeros
  S$=Str$(SCORE(H))-"  "  :  L=Len(S$)
  If  L<N  Then S$=String$("0",N-L)+S$
  HI_SCORE$(H)=S$+" "+NAME$(H) : Rem And add it to the name
Next H
Do
  Sort HI_SCORE$(0) : Rem Sort them in ASCENDING order
```

```
  Rem And display the in reverse order
  Locate 0,2
  For S=11 To 2 Step -1
   Cline : Print HI_SCORE$(S)
  Next S
  Print : Print "Enter your own name and score "
  Cline : Input "Name?";NAME$(11) : Cline : Input "Score?";SCORE(11)
  Rem pad out score
  S$=Str$(SCORE(11))-"  " : L=Len(S$)
  If L<N Then S$=String$("0",N-L)+S$
  HI_SCORE$(0)=S$+" "+NAME$(11)
  Print
Loop
```

Final Words

During this section, I've attempted to provide you with all the information you need to write fast-action arcade games in AMOS Basic. In fact, AMOS provides you with power to spare! So there's nothing stopping you from creating some terrific programs! Blast away!

```
Rem drunkards walk
Rem define some arrays for the missile and object data
Dim PMISS(7),ACTIVE(32),A$(2)
Curs Off : Flash Off : Fade 5
Rem load some aliens from the extras disc
Load "Extras:Sprite_600/aliens/alien4.abk"
Load "Extras:Sprite_600/space/ship1.abk",1
Rem set up the screen
Flash Off : Curs Off : Cls 0 : Gosub STARS : Hide
Double Buffer : Screen Display 0,128,70,,
Fade 5 To -1 : Wait 15*5
Rem set up the number and types of objects
SHIP=20 : MISS=9 : ALIEN=1
MFIRST=2 : MLAST=4 : AFIRST=5
Rem centre the hot spot on the ship, move it to the top centre of missile
Hot Spot SHIP,$11 : Hot Spot MISS,$10
Rem set up UCS
JDX=4 : JDY=4 : TX=0 : TY=128 : BX=320 : BY=192
Limit Mouse X Hard(TX),Y Hard(TY) To X Hard(BX),Y Hard(BY)
BY=BY-24
Rem define animation strings. These can be anything you wish
Rem But if you're positioning the aliens using a launch system
rem You'll need to make a few minor changes to the SET_LEVEL routine
Rem Anyway, I've used the RANDOM Walk system
A$(1)="Let R2=RA*2-1; Let R3=RA-1; Let R4=RA*8-1; Let R5=RA*4-1;"
A$(1)=A$(1)+"Horiz: Let R0=Z(R2)*8-R4;Let RX=X+R0;"
A$(1)=A$(1)+"If RX<0 then Jump Horiz;"
A$(1)=A$(1)+"If RX>320 then Jump Horiz;"
A$(1)=A$(1)+"Vert: Let R1=Z(R3)*8-R5; Let RY=Y+R1;"
A$(1)=A$(1)+"If RY<0 then Jump Vert"
A$(1)=A$(1)+"If RY>200 then Jump Vert"
A$(1)=A$(1)+"Move R0,R1,16;"
A$(1)=A$(1)+"Jump Horiz;"
Rem Here's a sequence to handle the explosions
AD$="For R0=9 To 15; Let A=R0; P; Next R0;"
AD$=AD$+"Let X=-100; Let Y=0;"
A$(2)="Loop:"
A$(2)=A$(2)+"Let R2=RA*2-1; Let R3=RA-1; Let R4=RA*8-1; Let R5=RA*4-1;"
A$(2)=A$(2)+"Let RF=Z(3); If RF<2 then Jump Horiz; else Jump Vert"
```

```
A$(2)=A$(2)+"Horiz: Let R0=Z(R2)*8-R4; Let R2=X+R0;"
A$(2)=A$(2)+"If R2<0 then Jump Horiz;"
A$(2)=A$(2)+"If R2>320 then Jump Horiz;"
A$(2)=A$(2)+"If R4=0 then Jump A; "
A$(2)=A$(2)+"Let RX=R2;"
A$(2)=A$(2)+"A:Move R0,0,16;"
A$(2)=A$(2)+"Vert: Let R1=Z(R3)*8-R5; Let R3=Y+R1;"
A$(2)=A$(2)+"If R3<0 then Jump Vert"
A$(2)=A$(2)+"If R3>200 then Jump Vert"
A$(2)=A$(2)+"If R4=0 then Jump B"
A$(2)=A$(2)+"Let RY=R3;"
A$(2)=A$(2)+"B:Move 0,R1,16;"
A$(2)=A$(2)+"Jump Loop;"
Rem set up AMAL channels
Channel 1 To Bob 1
For C=MFIRST To MLAST
   Channel C To Bob C
   Bob C,-100,-100,MISS
Next C
Rem Set up ship explosion
SHIP_EXP$="Anim 1,(40,1)(42,1)(43,1)(44,1)(45,1)(46,1)(47,1)(48,1)(49,1)"
LEVEL=0
Rem draw screen and load up the first level
Gosub SET_TITLE
Gosub SET_LEVEL
Rem Play game
Repeat
   Repeat
      Rem read joystick or mouse
      Gosub UCS
      Rem Check for a collision between the ship and an alien
      If Bob Col(1,AFIRST To ALAST) Then Gosub DEATH
      Rem Check for a collision between the aliens and a missile
      Gosub DESTRUCTION
      Rem Fire missile
      If Mouse Key or Fire(1)
         Gosub MISSILE
      End If
   Rem Until all aliens are dead or the ship has been destroyed
   Until HIT=NALIENS or LOST
   Rem If the player has been the current level, load a new one
   If LOST=0
      HIT=0 : Inc LEVEL : Gosub SET_LEVEL
   End If
Until LOST
Stop
DEATH:
Rem Handle the destruction of the players ship
Boom : Amal Off : Amal 1,SHIP_EXP$ : Amal On 1
Wait 50 : LOST=1 : Return
DESTRUCTION:
rem check for the death or an attacker
DEAD=0
Rem check each missile in turn
For M=MFIRST To MLAST
   If Bob Col(M,AFIRST To ALAST)
      Rem Check each alien
      For A=AFIRST To ALAST
         Rem ACTIVE(A) is set to 0 when the aliens killed
         Rem So this line checks for each active alien
```

```
         If Col(A) and ACTIVE(A) : DEAD=A : End If
      Next A
   End If
Next M
Rem If an aliens dead, call up an explosion
If DEAD Then Gosub DIE
Return
DIE:
Rem Destroy an alien and update the score
SC=SC+100 : Inc HIT : ACTIVE(DEAD)=0
Screen 1 : Locate 34,1 : Print SC; : Screen 0
Amal DEAD,AD$ : Amal On DEAD : Boom
Return
MISSILE:
REm handle multiple missiles
If Timer<TM+12 Then Return
MN=1 : Gosub ALLOCATE : If ALL=0 Then Return
Bob PMISS(0),X Bob(1),Y Bob(1)-30,MISS
Amal PMISS(0),"Move 0,-210,60" : Amal On PMISS(0) : Shoot : TM=Timer
Return
Rem allocate a free missile
ALLOCATE:
ALL=0
For M=MFIRST To MLAST
   If Chanmv(M)=0
      PMISS(ALL)=M
      Inc ALL
   End If
   Exit If ALL=MN
Next M
Return
Rem Set up each level
SET_LEVEL:
Rem Display number
Wait 100 : Screen 1 : Locate 10,1 : Print LEVEL; : Screen 0
Rem Choose the number of aliens
NALIENS=(LEVEL mod 8)+1
Rem Set the type of attack wave. You could use different images
Rem for each wave and load them with LOAD..
TYPE=(LEVEL/8)
ALAST=AFIRST+NALIENS-1
Rem Switch off all AMAL channels
For D=ALAST+1 To AFIRST+7 : Amal Off D : Next D
Rem position each alien
For C=AFIRST To ALAST
   Channel C To Bob C
   Bob C,Rnd(320),Rnd(80),ALIEN
   ACTIVE(C)=1
   Rem start attack
   Rem Switch attack patterns every eight levels
   If TYPE mod 2=0
      Amal C,A$(2)
   Else
      Amal C,A$(1)
   End If
Next C
Rem Increase speed every sixteen levels
SPEED=TYPE+1
Amreg(0)=SPEED*4-1
Amal On
```

```
Return
SET_TITLE:
Rem Display a simple title line
Screen Open 1,320,16,2,Lowres
Curs Off : Paper 0 : Colour 1,$F : Pen 1 : Cls
Rem Create rainbow effect
Set Rainbow 1,1,16,"","",""
For R=0 To 8 : Rain(1,R)=R*$22 : Next R
Rainbow 1,0,49,9
Locate 0,0 : Centre "Drunkards Walk"
Locate 4,1 : Print "Level "; : Locate 28,1 : Print "Score "; : Screen 0
Return
UCS:
Rem Read mouse or joystick
If Joy(1)
   If Jleft(1) and(SX>TX) : Add SX,-JDX : End If
   If Jright(1) and(SX<BX) : Add SX,JDX : End If
   If Jup(1) and(SY>TY) : Add SY,-JDY : End If
   If Jdown(1) and(SY<BY) : Add SY,JDY : End If
   X Mouse=X Hard(SX) : Y Mouse=Y Hard(SY)
Else
   SX=X Screen(X Mouse) : SY=Y Screen(Y Mouse)
End If
Bob 1,SX,SY,SHIP
Return
STARS:
Rem Generate starfield
For I=1 To 100
   Ink Rnd(13)+2 : Plot Rnd(320),Rnd(200)
Next I
Return
```

4

Adventure Games

4.1 Introduction

4.1.1 A little history

All modern adventure games are descended directly from a single Fortran program written in the early 70s. This was the infamous Advent game by Crowther and Woods. When it was first produced, Advent was one of the largest computer programs ever written. But computers have come a long way since 1970. Amazingly, it's now perfectly feasible to fit the entire game onto a standard A500!

As you can well imagine, the games software market for an early 70s mainframe was just a little limited. So Advent was never marketed commercially, and poor old Crowther and Woods didn't made a penny out of their idea. They did however, ensure themselves a respected place in the history of computer games. In fact, the first truly commercial adventures were in the Scott Adams series released around 1979. These managed to enthral a whole generation of computer enthusiasts. For me though, the modern adventure began with a piece of artificial intelligence research.

The Zork program was originally designed as a effective demonstration of the way computer systems could be made to ***understand*** simple English sentences. But Zork had all the characteristics we now associate with an adventure, including the ability to handle genuine English instructions. There was none of this boring two word *Attack Alien* stuff. We could now go overboard, and enter our commands in full typing something sensible like *Attack the giant alien with the yellow banana.* A stunning improvement!

Subsequently, some of the programmers behind this project decided to try to produce a version of this game for use on a home computer. That's how the Infocom company was born. Eventually of course, even Infocom were forced to admit defeat. The final program had to be split up into three complete adventures in order to fit onto the original microcomputer systems. It's impossible to over-emphasise the impact Infocom have had on the adventuring scene. So it's sad to have to report their recent demise as an independent company. Thankfully, the best of the Infocom adventures are still readily available, at bargain basement prices (£10). The current range includes classics like Zork, Planetfall, and the amazingly funny Hitch-hiker's Guide to the Galaxy adventure. Great stuff!

The fall of Infocom can be directly attributed to the rise of the so called graphic adventure. As computers have evolved, it's now become possible to supplement our text descriptions with mind-blowing screen effects. In Britain, companies like Magnetic Scrolls have lead the way with innovative adventure games such as The Pawn and Wonderland. And in the USA, Sierra On-Line have been breaking new ground with the infamous Leisure Suit Larry games.

Suffice it to say, the effects of these developments have been startling! The text adventure has been almost completely superseded by games which seamlessly combine text, pictures and sound effects to generate a realistic imaginary world for the player to experience. Intriguingly, the Amiga is at the forefront of these developments. Commodore are currently working on a whole new generation of interactive video systems based on the Amiga. Within the next few years, CD storage devices will almost certainly become a familiar part of the Amiga computing scene. Since these systems are tailor-made for the creation of adventure games, there's an exciting future ahead for all you budding adventurers out there!

4.1.2 Adventurers start here

In a few moments, I'll be demonstrating how you can create one of these adventure games yourself. But what actually is an adventure? Well, the game takes place in an imaginary world in which practically anything is possible. The player takes the part of an heroic inhabitant of this world, embarking on an exciting (and probably suicidal) quest. It's the player's job to guide the character using a range of simple English commands. Depending on the quality of our advice, the adventurer will either solve the puzzles and complete the game, or experience an untimely (and probably agonising) death. The underlying appeal of adventures is that they allow us to temporarily abandon the frustrations of the real world, and embark on a realistic and satisfying journey into the realms of imagination.

The easiest way to familiarise yourself with an adventure, is to play one. The AMOS Data disk contains a small example program in the file CASTLE_AMOS.AMOS. Castle AMOS isn't a complete adventure by any means. So if you're already familiar with this type of game, you'll probably be able to solve it in a matter of minutes. Despite its simplicity though, it does provide a perfect demonstration of the type of adventuring system that can be created using AMOS Basic. It also incorporates some terrifying digitised pictures of a few of the AMOS development team! (Not me, of course. We weren't that sadistic!)

If you want to sample a real adventure, it's well worth checking out your favourite PD Library. There are dozens of playable adventure games in the public domain, including a complete version of the original Advent program. Over the years, it has been revised and rewritten several times. The latest version is known as Colossal Cave.

4.1.3 Types of adventure

The first decision we will be faced with is to decide on the basic format of our game. This is important because the sheer power of the AMOS programming system allows us to create anything from a simple text adventure to a full blown graphic one.

Here's a brief description of the two most common adventuring systems.

Text-plus adventures

These games are the direct descendants of the original text adventure. The players type their commands in standard English, directly from the keyboard. The computer then quickly works out the results, and displays the characters new predicament on the screen, with an inspiring piece of descriptive text.

Unlike the Colossal Cave adventure, the description is freely supplemented by pictures, sound effects and even the occasional snatch of music. But these effects are used solely to add extra atmosphere to the game. They play absolutely no part in the various puzzles. Although these games are extremely unfashionable, text-plus adventures still have their adherents, and there are plenty of excellent examples on the market. Look out for programs such as The Pawn or Fish from Magnetic Scrolls.

Graphic adventures

Graphic adventures describe the player's surroundings using a combination of text and graphics. Each location in the game is assigned its own individual picture. What's more, all the objects encountered by the player are represented on the screen using their own individual icons. Icons are small pictures which can be selected with the mouse pointer, usually by pressing the left mouse button. If you've played around with the AMOS sprite editor, you'll have already encountered them, as they're used for the various buttons in the control panel.

In practice, even the most detailed picture can be pepped up with a little bit of text. So many graphic adventures also provide a separate text window. The character is manipulated using either standard English, or from a series of icons which can be selected directly with the mouse. Normally, these buttons are arranged in the form of a separate control panel on the screen. This provides immediate access to common commands such as movement, and speeds up the gameplay considerably.

AMOS Basic contains a number instructions which allow us to generate these control panels ourselves. The only difficult part is designing all the buttons. This Icon system is so amazingly powerful in fact, that it can completely replace the text entry routines. Personally, I find this approach robs an adventure of much of its appeal. If we list all the possible commands in advance, we automatically make the solution of the adventure considerably easier.

There's a lot of fun to be had from searching for the correct combination of words required to solve the current puzzle. Obviously it all boils down to a matter of taste. But even if you're a die-hard graphics enthusiast, it's well worth looking at the text analysis system I'll be showing you in Section 4.8.

4.2 Scenario Design

4.2.1 Descriptions

If we want to produce a truly playable adventure, it's vital to generate a believable sense of atmosphere. The gameplay is centred totally on the imaginary world we have created. We must therefore go to considerable lengths to keep the action as exciting as possible.

In a pure text adventure, we can create this atmosphere by packing the room descriptions with lots of adjectives. So a tree would be described as:

A massive oak tree blossoming with the generous bounty of spring

rather than the more prosaic, but equally accurate:

A tall tree.

It's important not to get carried away with this idea though. If we allow our descriptions to become too flowery, we'll probably just send our players to sleep!

4.2.2 Mocking up the pictures

Graphic adventures obviously require detailed pictures for each individual location. During the initial planning stages, it's best to limit ourselves to simple sketches. These can be quickly entered into the computer using a drawing package such as Deluxe Paint, and can be incorporated directly into the early versions of our adventure. We can now test our scenario ideas carefully, before we commit ourselves to the final artwork. Otherwise we could expend a great deal of time and effort on scenes which are never actually used in our game.

4.2.3 Creating a map

The first step is draw up a complete map of our game on paper. This map should include full details of the connections between the various rooms, along with any ideas we may have about the expected contents.

After we've created our maps, we can eagerly get to work inventing the puzzles and traps which will be encountered in the adventure. For me, this activity is one of the most enjoyable parts of the entire process. As we devise our puzzles, we should carefully mesh them into the background of our game world. Games which lack any form of consistent pattern, no matter how twisted or bizarre, invariably fail to engender the believable atmosphere which is so vital to a successful adventure.

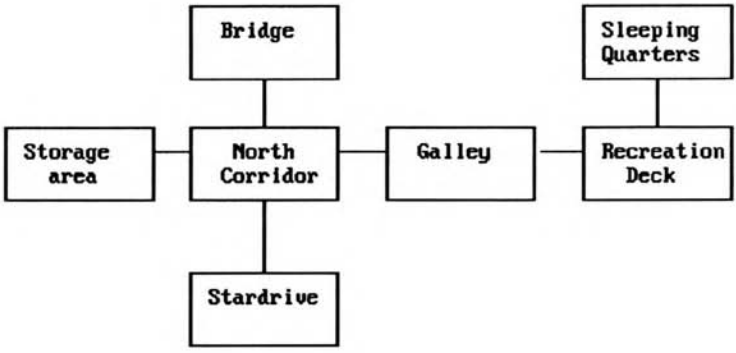

Figure 4.1: Section of a game map

Sometimes, this is just common sense. If we've set our adventure in a world of swords and sorcery, it's clearly pointless to have problems requiring the player to fix a computer! But in other cases, it's rather more subtle. Remember, the player is supposed to be controlling a real character, and we need to ensure that the player totally identifies with this person.

Supposing for instance, the player is controlling an introverted scientist type. If we were to ask this character to fight an alien with his bare hands, he'd run away in horror! Similarly, a Rambo style adventurer would find it very difficult to endure complex negotiations with an evil dictator. He'd simply blast away regardless!

If we continually expect our characters to act contrary to their basic instincts, the players just won't believe in them, and the adventure will be ruined! So we have to set our puzzles according to the abilities and personality of the character who is supposed to be experiencing them.

The best adventures are open ended, and the best puzzles have several possible solutions. We should therefore spend quite a bit of time generating various alternative ways of winning the game. There's nothing more annoying for the player than coming up with a perfectly sensible solution, and having it rejected out of hand, just because the designer didn't think of it. So it's essential to include all the obvious solutions to our puzzles directly in our program. We can't of course, cover every possibility, but the more variety we can add to our game, the deeper and more engrossing it will be for the player.

In my experience, the most enjoyable puzzles have solutions which appear to be blindingly obvious in hindsight, but which are fiendishly difficult at the time. Remember, nobody enjoys a puzzle which is too easy or ridiculously contrived. But if we can get the players to regularly kick themselves for their stupidity our scenario will be providing them with a real and exciting challenge. A typical example of this technique is to subtly alter a standard message to include some vital piece of information. Completely different locations can be given almost identical descriptions, with just a few crucial differences. It's amazing how often players will just skim through the text without actually reading it. And once they finally realise the solution, they will be amazed to discover that the information has been in front of their eyes all the time!

We can also attempt to describe a commonplace object in an unfamiliar way. In a world of magic, we can have great fun adding mechanical devices such as cars or refrigerators. These can be explained through the eyes of the character experiencing the adventure, without reference to any of the assumptions of our modern day world.

So a car might be regarded as a gigantic metallic monster with a horrible teeth and a terrifying roar. Imagine the chaos which would ensue if our adventurer attempted to rescue an unfortunate damsel who was being *eaten* by this monster. Rather than actually being grateful, the girl would be extremely upset at being dragged out of her car by some moronic barbarian with delusions of grandeur!

As our story opens, it's a good idea to start off with something relatively simple. We can then slowly increase the complexity of the problems during the course of the game. This gently draws the players into our world, without frustrating them unnecessarily at the beginning.

4.2.4 Designing the graphics

Like it or not, the original text adventure is now almost completely dead. So even die-hard Infocom freaks have been grudgingly forced to include some sort of graphics in their adventures. Although this has undoubtedly led to a dramatic improvement in the quality of commercial games, it's also posed a real problem to the budding adventure creator. Many adventures now include terrific graphics from the hands of top-rate professional artists. How can mere mortals such as ourselves possibly compete with these immensely talented people? Well, there are several solutions.

Collaboration

The easiest way of generating our graphics is to harness the artistic abilities of a friend or relation. If we're lucky, we may be able to find someone who would be delighted to help us out.

Capturing the images from an external source

If all else fails though, I'm afraid we'll have to do the whole job ourselves. This may seem a pretty tall order, especially if you are a colour-blind, artistic incompetent like me! Fortunately, all is not lost. With the help of some extra hardware, we can capture our images directly from an existing photograph or a video. Once we've loaded these pictures into our Amiga, we'll be

able to edit them straight from a drawing package such as Deluxe Paint II. Believe me, this is much easier than drawing the graphics from scratch!

It's worth emphasising however, that all images used in your games should be from a totally legitimate source. Never attempt to *borrow* images from an existing film, magazine or program. This is illegal! The same copyright rule also applies even if we're only intending to distribute our game in the public domain. Having said that, the chances of getting caught red-handed are extremely slim. So it all boils down to a question of morality, which I'll wisely leave to you.

Hand held scanners

These are small mouse-like devices which plug into the back of our Amiga and allow us to grab our images directly from any black and white photograph. Armed with a simple still camera, and a few basic props, we'll be able to create detailed pictures of anything from a castle to a space-ship.

Hand held scanners can be acquired for around £100, and are capable of generating some quite reasonable screen effects. On the minus side though, they're not exactly the easiest things to use. You'll therefore need to indulge in a lot of experimentation before you can achieve anything like acceptable results. If possible, try before you buy!

Video digitisers

Digitisers load an image straight from a standard video camera, and are now used extensively by many commercial software houses. A good example of the type of images which can be generated by these systems can be found in Castle AMOS. This incorporates digitised pictures of some of the AMOS development team (including Francois Lionette himself!).

Generally, digitisers are easier to use than scanners, and are capable of producing excellent images with comparatively modest amounts of hardware. Good results can be obtained from systems costing between £150 and £200, although you should expect to pay more for extras such as colour.

In addition to the digitiser, you'll also need access to a video camera and a tripod. Cheap monochrome video cameras are now available for approximately £200, but this brings the total cost up to £400! So unless you can borrow a video camera, these systems are probably just an impossible dream. There are however, several companies which will take our images and digitise them for us professionally. Look out for details of these services in the advertising sections of your favourite computer magazine.

4.3 Basic Concepts

Once we've created our map and designed the various rooms, we'll be ready to begin work on the actual game. The overall mechanics of an adventure can best be summarised with a little pseudo-code as shown on the next page, in Figure 4.2.

Before you can make sense of this description, you'll need to be introduced to some new terms.

Events include all the possible occurrences in our adventure – things such a door opening, a monster appearing, or even one of our characters legs falling off for no readily apparent reason.

A **high priority event** is one which must be performed immediately, such as the character's untimely death. Whenever this occurs, the program should terminate at once, just like the character. There's no need to bother to read the keyboard, or display the room description. The character is DEAD! High priority events are rather like the *Go to jail* cards in Monopoly, as they take complete precedence of any other commands in the game.

Low priority events are slightly less earth shattering in nature. They deal with things like hunger, which can often be avoided with a little action on the part of the player (Eating some food would certainly help!).

Local events are situations that can only arise if the character is occupying a particular location. Traps, for instance, will only be sprung when the character enters a certain room.

```
repeat
perform high priority event
describe room
input a command from the user
test for valid commands
if system command then interpret system command
interpret global commands
if movement command
then
  if move is allowed
  then
    enter new room
  endif
  endif
interpret local commands
perform local events
perform low priority events
until player exits,wins, or loses
```

Figure 4.2: Pseudo-code description of an adventure

We can also use a similar system to divide up the player's instructions:

Movement commands are instructions to the character to move in a certain direction. They include phrases like: GO EAST, WEST, ENTER SHIP and so on. In a graphic adventure movement is normally controlled using a series of Arrow icons on the screen.

System commands are used to control the actions of the adventure program itself and include instructions such as QUIT, SAVE or INVENTORY.

Finally, there are the **local** and **global** commands. Local commands can only be performed at a specific location, whereas Global commands can be executed anywhere in the adventure.

So OPEN WINDOW is a local command because it can only take place in the room with the window. But EAT FOOD is global because it can be entered whenever the character is feeling a little peckish. Here are a few further examples of these commands:

System	Local	Global
RAMSAVE	PRESS SWITCH	GET
OOPS	CLIMB WALL	DROP

4.3.1 Standard routines

If you examine carefully the pseudo-code in Figure 4.2, you should be able to isolate a number of the key components in an adventure game. For the sake of clarity, I've grouped these routines into two separate lists.

Global routines
perform high priority event
perform low priority events
input a command from the user
test for valid commands
interpret system commands
interpret global commands

These are general purpose routines which are used throughout the adventure. They are can be implemented using either procedures or subroutines (GOSUBS).

Local routines

describe room
interpret movement commands
interpret local commands
perform local events

The action of these routines will obviously vary from room to room. So each room in the adventure will need its own individual versions. We can accomplish this by placing all the instructions concerning a particular room into a single *handle room* routine.

You might feel tempted to use procedures for this purpose, and create a separate procedure for each room in the adventure. Unfortunately, this makes it extremely difficult to move between the various rooms, as there's a definite limit to the number of procedure calls we can nest inside each other.

If all that sounds insanely complicated, rest easy. The system I'll be using in this chapter is far simpler! Instead of using procedures, we'll place each location directly inside our main program, and jump from room to room with a simple GOTO statement. Here's a program fragment which illustrates this technique:

```
Rem This is just a FRAGMENT. Don't type it in!
Rem Start of program
Gosub INITIALISE_GAME : rem Call main initialisation routine
ROOM1: rem Handle everything required for room 1
R=1:Rem Set up room number
Gosub HI : Rem call high  priority event
Rem Describe room
Rem See section 4.7
If SHORT
  Print "Small Room"
Else
  Rem load a picture from the disk in screen 1 (optional)
  Load Iff "Picture1",1
  Print "You are in a small dark room. A sign on the wall says "
  Print "Location number one. There is a massive iron door to the north"
Endif

Rem Call a series of routines to read the commands from the user
Rem and handle any global commands or events
Rem I've included them here just for completeness
Rem But they'd normally be placed in a separate routine
Rem and called with something simple like: GOSUB USER
Gosub ENTER : Rem input a command from the user
Gosub CHECK : Rem test for valid commands
Gosub SYS : Rem handle system commands
Gosub CHECK_GLOBAL : Rem interpret global commands
rem handle movement
rem D contains movement direction.
rem It's set to zero if player didn't enter a movement command
If D=EAST Then Goto ROOM2
  : :
  : :
Rem do local command
If ACTION=OPEN and ITEM=DOOR
  Print "The door slowly creaks open":DOOR_OPEN=True
Endif
```

```
Rem Local events go here
  : :
  : :

Rem Handle low priority events
Gosub LO
Rem Jump back to start of room routine
Goto ROOM1

ROOM2: Rem Next room : :
  : :

LAST_ROOM: Rem The label used for the room can be anything you like
  : :
  : :

End

Rem Global routines go here
  : :
  : :
```

Don't worry if you don't understand everything in the above example immediately. I'll be carefully explaining each of the components separately during the remainder of this chapter. It's only intended to give you a general idea of the basic structure of the adventure game.

4.4 Describing the Rooms

Every room in the adventure needs to be lovingly portrayed in a considerable amount of detail. In a text adventure, each room will have two separate descriptions. The long description will be displayed the first time the player enters the room, and whenever the LOOK or DESCRIBE commands are entered. There will also be a shorter description which will be printed on any subsequent visits.

4.4.1 The long room description

This is a complete word picture of the current room which includes vital information needed to solve the adventure. It is essential to provide the players with enough information to completely visualise the room, as this will encourage the feeling that they are actually experiencing the adventure. As I mentioned earlier, colour is often added by packing the description with plenty of adjectives and adverbs.

Look at the following portrayal of two identical rooms.

You are in a thirty feet high chamber. There are exits to the east via a canyon and to the west through a small passage. A bird is singing in a corner.

You are in a splendid chamber thirty feet high. The walls are frozen rivers of orange stone. An awkward canyon and a good passage exit from the east and west sides of the chamber. A cheerful little bird is sitting there singing.

The second piece of text is taken directly from the original Advent program. Maybe the English is a little rough, but there's no doubt about which of the two descriptions is the more atmospheric.

Another thing to consider is humour. Used sparingly, this can add a nice element of fun to the adventure. But don't be too heavy handed. Repeated and unnecessary sarcasm can get very irritating after a while, and there's nothing so unfunny as a really bad joke.

4.4.2 The short room description

This provides a one line description of the present location, and is used to briefly portray a room which the player has already encountered. In the previous example the short description would be just:

Splendid Chamber

The short description speeds up the gameplay by limiting the amount of information presented to the player during the game. If the player requires the full description, it's usually accessible by typing LOOK or DESCRIBE from the keyboard. Some games also allow the player to insist on a long description whenever the character enters a room. The original Infocom games provided a special VERBOSE command for this purpose.

4.4.3 The graphics

Graphics are now an established part of the modern adventure game, so even the simplest text-plus systems require detailed location displays. If we're writing a graphic adventure, the design of these images is vitally important.

The key to the creation of effective graphics is to generate a consistent and realistic sense of atmosphere. In practice this is extremely difficult, and that's one of the reasons why companies such as Infocom were so reluctant to include them in their games.

When we're creating these graphics, we'll also have to spare a thought about the amount of memory they will consume. Luckily for us, Text-plus adventures only require us to generate a limited number of pictures. With the help of the AMOS Basic compaction instructions, we can easily cram 20 full-sized 16-colour images onto a single disk. These can be loaded into our game as and when they are needed. So we're unlikely to encounter any serious memory problems at all.

In a graphic adventure though, we'll need a separate picture for each location. And there's no way we'll be able to compress a hundred or more locations directly on the disk. But when the going gets tough, it's time to get sneaky!

Here are a few ideas:

❑ Restrict all our graphics to a small window on the screen

This will leave plenty of room for the various icons and text displays. A good compromise is to limit our images to around half the total screen area, as we'll now be able to compress each picture into less than 16k. We'll therefore be able to hold more than 50 different locations on a standard 3 1/2 inch disk.

❑ Replace commonly used images by a single picture

Most adventure games include a number of similar locations such as corridors, entrances and stairways. If we attempted to create a separate image for each room, we'd obviously waste a great deal of valuable memory. We can however, replace many of these descriptions by a single generic picture.

We don't need to make every corridor look identical though! Each location can be customised by pasting extra screen elements at the appropriate points on the display. We could take these

elements directly from the sprite or icon banks, and compact them still further using the AMOS Squash utility.

Supposing we wanted to create several pictures of a simple door. We might store a large selection of handles, frames and signs in the icon bank. These could be pasted into position with a series of PASTE ICON commands, allowing us to generate dozens of different doors in just a fraction of the normal memory. Note that I'll be discussing the icon bank in detail in Chapter 7. But since the PASTE ICON command is very simple, here's a quick description:

PASTE ICON x,y,i

This draws icon *i* at screen coordinates *x,y*.

Our icons can be created in two main ways. One possibility, is to draw them as standard 16-colour Bobs using the AMOS sprite editor, and then convert the resulting image files to icon format with the ICON_TO_BOB.AMOS utility. Alternatively, if we're already using the latest Spritex editor, we can save our icon images directly in the correct format. If you haven't already got it, Spritex is available separately from the AMOS User group.

4.4.4 Storing the rooms

Ok, so we've created the descriptions for each room, and drawn the various pictures. The next job is to enter them into the Amiga's memory. Each room will need to be assigned its own individual identification number. Normally, this value will be set at the start of each location routine using a line like:

```
R=1
```

We'll also need some way of detecting whether the player has already visited the current room. This can be accomplished with the help of a VISIT array. Each element in this array corresponds to a single room in our adventure. Whenever a room is entered, the appropriate element can be increased by one. Our program can now use this value to automatically choose between the long and short text descriptions. For example:

```
Rem This may look odd, but there's a trick involved
If VISIT(R)>0 and VERBOSE<>0 Then SHORT=1 Else SHORT=0
If SHORT
  Print SHORT$
Else
  Print LONG$
Endif
```

You may wondering about the odd way I've split the tests in two parts. Wouldn't it have been faster to have incorporated the whole thing into a single If..Else..Endif statement?

```
If VISIT(R)>0 and VERBOSE<>0
  Print SHORT$
Else
  Print LONG$
Endif
```

However, each ROOM routine will have its own separate version of these commands. Since we'll be retyping these instructions whenever we add a new room to our adventure, it's sensible to keep them as short and simple as possible. There's obviously no point in making life difficult for ourselves.

Although the second routine is shorter, the first is easier to use. That's because the `If VISIT(R)>0....` bit can now be safely tucked away in our *high priority* routine. This allows us to chop half a line or so out of each room handler. What's more, if there are a lot of rooms, we can replace the variable *SHORT* by just *S*. So the display commands would boil down to:

```
If S
  Print SHORT$
Else
  Print LONG$
Endif
```

As with many aspects of programming, there are literally dozens of possible storage strategies. Here are a couple of ideas.

4.4.5 Storing the description on the disk

The Amiga's disk system may be a little slow, but it's ideal for storing our graphics and sound effects. All our images can be placed in separate files on the disk, and can be quickly loaded into memory when required. Usually, the graphics would be created using an external drawing package such as Deluxe Paint. We'd then compact each picture with a call to either the AMOS Basic SPACK or PACK commands. The choice between these commands largely depends on the type of the adventure we are creating.

`SPACK s To b`

Compresses screen *s* into memory bank number *b*.

`SPACK s To b,tx,ty,bx,by`

Packs the rectangular area from *tx,ty* to *bx,by* into the appropriate memory bank. SPACK is best for text-plus adventures. When we display a SPACKED image, the picture winks cleanly into view, and there's no unpleasant flicker effect. Alternatively, if we're creating a graphics adventure, we'll want to use the PACK command instead. The format's identical to SPACK.

`PACK s To b`

Packs screen *s* into memory bank number *b*.

`PACK s To b,tx,ty,bx,by`

Compresses just a section of the screen. PACK is designed especially for animation, and therefore requires a double buffered screen in order to work. Unlike the SPACK instruction though, it doesn't save the colours and position of the current screen along with the image.

After we've packed our images, we can display our pictures at the appropriate points in our program with the UNPACK command. There are separate versions of this instruction for SPACKED or PACKED images.

`UNPACK bank To screen`

Restores a SPACKed screen to its original position.

`UNPACK bank`

Displays a complete PACKED image on the current screen. This screen must have the same colour assignments as the original or the result will look extremely weird! Warning: If the current screen is smaller than our compressed image we'll get a nasty error message!

UNPACK bank,x,y

Draws a PACKED image at coordinates *x,y*. We could use these commands to display our image at the start of our room routine like so:

```
ROOM1:
Gosub HI
If SHORT
  Print "Short bit"
Else
  Rem Load picture of room into a memory bank
  Load "image.abk",4:rem load image.abk from disk and place in bank 4
  rem Unpack image and display it onto the screen
  Unpack 4 to 1: rem Unpack SPACKED image from bank 4 into screen 1
  rem For images created with PACK we'd use something like Unpack 4,0,0
Endif
: :
rest of routine goes here!
```

Now for a complete example of the PACK and UNPACK commands in action.

Example 4.1: PACK and UNPACK in action

```
Rem AMOS picture compressor
Screen Open 1,640,16,4,Hires
Curs Off : Cls 0 : Paper 0
Centre "<AMOS Picture compressor>"
Rem Main Loop
Do
  Screen 0
  F$=Fsel$("","","LOAD A PICTURE TO COMPRESS")
  Exit If F$=""
  Load Iff F$,0 : Rem load picture in screen 0
  Rem use SPACK 0 To 4 for a TEXT plus adventure
  Pack 0 To 4 : Rem pack entire screen into bank 4
  Rem use PACK 0 To 4,TX,TY,BX,BY to pack part of screen
  Rem TX,TY=Coords of top left corner of area.
  Rem BX,BY=Coords of bottom right corner
  Rem SPACK can also be used in exactly the same way
  Screen 1
  Print "The length of your picture is ";Length(4);" bytes "
  Wait 50
  Screen 0
  F$=Fsel$("*.abk",F$,"SAVE A COMPRESSED PICTURE")
  Exit If F$=""
  Save F$,4
Loop
Erase 4
```

This program loads a series of pictures and compresses them onto the disk. Note that any pictures can be used for this purpose, including those drawn in the amazing HAM or Half Brite modes! Sound effects or Music can be loaded straight off the disk as and when they are required. A typical routine would be:

```
Load "AMOS_DATA:Music/Musicdemo.abk"
Music 1
Load "AMOS_DATA:Samples/Samples.abk"
Samplay 1
```

As you might expect, we can also store text descriptions on disk. Frankly, this is only worth considering if our descriptions are extremely long, or if we're running short of memory. That's because it will require us to create a separate database on the disk using random access filing techniques. Here's a run-down of the general procedure.

- ❑ The first step is to create a separate program to enter our text onto the disk.
- ❑ We can then decide on the maximum length of each location description. A good compromise is to use 10 lines of 40 characters. That's about 400 bytes per record.
- ❑ Next, we open a random file to hold all the text in our adventure. This can be created with:

```
Open Random 1,"Advent.txt":rem use any filename you wish
```

- ❑ Before we can save our text into this file, we'll need to define the various fields to hold each room description. We'll call up the AMOS Basic FIELD command like so:

```
Field 1,40 as S$,400 as L$
```

- ❑ Now we take each location in our adventure in turn and place the short description into *S$* and the long version into *L$*. We can then save this description onto the disk using a line such as:

```
Put 1,room_number
```

Each description will be defined with a section of code like:

```
ROOM1:
S$="Short description"
L$="This is part of the long description and contains all your"
L$=L$+"text for the current location"
R=1: Rem set room number to location N
Put 1,R
ROOM2:
: :

etc
```

- ❑ At the end of our program, we should close the random file with:

```
Close 1
```

- ❑ The penultimate step is to save our new program onto the disk, so we can edit it at a future date. This will allow us to make changes to our adventure during the development process.
- ❑ Finally, we run our new program to generate our data file.

We'll be then able to access the room description inside our adventure with the following code:

```
Rem This is only a fragment. So don't type it in
rem Open our file
Open Random 1,"Advent.txt":rem Place at start of our adventure
  Field 1,40 as S$,400 as L$
: :
ROOM1: rem Handler for room 1
R=1
Gosub HI :Rem Hi priority events
Get 1,R: Load text for current room
If SHORT Then Print S$ else Print L$
: :
Interpret commands etc..
```

I warned you it was complicated, didn't I? Thankfully, after the file has been produced, each description can be loaded into memory extremely quickly, even from a floppy. Furthermore, since our text is held on the disk, it consumes practically no RAM. It's therefore possible to cram hundreds of pages of text into a standard A500 adventure. If you're interested, check out the *Disc Access* chapter from AMOS manual for details of the random access filing system.

4.4.6 Direct program display

A better approach is to display our text directly inside our room routine. This can be easily achieved using a series of PRINT or TEXT instructions. AMOS Basic provides us with dozens of powerful text commands which can be harnessed to add a whole new dimension to our text descriptions.

So these descriptions can be displayed anywhere on the screen, and can use any of the available fonts. Instead of just plain text, we can create realistic impressions of newspapers, tickets or even handwriting. These effects are surprisingly easy to produce and certainly add a genuine sense of atmosphere to our game. Here's an example.

```
Screen Open 0,640,200,16,Hires : Paper 0 : Curs Off : Cls 0
Rem This routine handles everything which happens in the dining room
Rem Don't type it in. It's not a complete program.
DINING_ROOM:
Rem HI: Rem call hi priority events
Rem Describe room
If SHORT
  Under On : Centre "<The Dining Room>" : Under Off
  Print : Print : Print
Else
  Print "You are in a large dining area, dominated by a massive ";
  Print Paper$(6);"blue";Paper$(0);" table"
  Print "A plaque on to top reads";
  Print : Print : Print
  Locate 30, : Print Pen$(5);Border$(" Food for Thought!",1);Pen$(1)
Endif
: :
Rest of routine goes here

Goto DINING_ROOM
```

BORDER$, by the way, is a neat little function which adds an attractive border around our text. It's compatible with either the PRINT or CENTRE commands. The format is just:

BORDER$(text$,b)

where *text$* is our message text, and *b* is a border number from 1 to 3.

One snag with BORDER$ though, is that it generates odd results if the text cursor is positioned at column zero. So we'll need move the start of our text one place to the right with a LOCATE or CRIGHT command before use as follows:.

```
cright: print border$("Hi there",1)
cright: print border$("Button",2)
cright: cls 0:print border$("test",30)
print border$("Whoops",1)
```

Also note PEN$() and PAPER$(). These are commands which allow us to change the colour of our text directly inside a PRINT instruction.

=PEN$(index)

Sets the colour of our letters to *index*

=PAPER$(index)

Changes the background colour of our text

```
print pen$(4);"Hi";pen$(5);"there"
print paper$(6);"Hello"
```

Check out the following instructions from the AMOS Manual.

UNDER ON/OFF

Underlines some text (But beware! It doesn't seem to compile properly!)

SHADE ON/OFF

Adds a nice shading effect to our text.

INVERSE ON/OFF

Swaps around the pen and paper colours.

WRITING

Changes the writing mode used to display the text.

CDOWN/CUP/CLEFT/CRIGHT

Moves the text cursor.

=AT(X,Y)

Positions the text cursor inside a string. This is quite similar to LOCATE.

We'd obviously need indulge in a fair amount of experimentation in order to create the precise effects we wanted. But once you've familiarised yourself with the instructions, it's all pretty easy.

In a graphic adventure, we can also use the PASTE ICON commands in a similar way. As I mentioned earlier, we could customise a single screen for use with several different locations. Each location display could be created by pasting extra objects at the appropriate coordinates. These could be held as normal images stored in the current icon bank.

The only snag with this system, is that it's extremely easy for the players to cheat, as they can simply list the program in AMOS Basic and read all the descriptions. This won't be a problem if we're intending to sell our game commercially, as we can simply convert it into machine code with the help of the AMOS Basic compiler. Once we've compiled a program in this way, it will be practically impossible for the user to obtain a listing. We'll also get a substantial speed increase! If however, we intend to distribute our programs with the RAMOS (Run only) system, we'll need to be able to conceal our descriptions directly inside our listings. Here are a couple of procedures to code and decode our text.

These procedures are not, of course, intended to foil the determined hacker. At best, they provide a useful measure of security to our games, without requiring massive amounts of extra programming.

Example 4.2

```
Rem test procedures
Do
  Input ">";E$
  ENCODE[E$]
  P$=Param$
  UNCODE[P$]
Loop
Stop
Procedure ENCODE[C$]
  Rem encodes a piece of text and prints out the result
  Rem Input string should be in normal English and should not include
  Rem control characters.
  Rem The PASSWORD can consist of any string of numbers you like
  PASSWORD$="123456"
  For C=1 To Len(C$)
  Rem Take each character from the password one by one
  PC=C mod Len(PASSWORD$)+1 : Rem I know it looks awful but it works!
  CODE=Asc(Mid$(PASSWORD$,PC,1))-Asc("0")
  V=Asc(Mid$(C$,C,1))
  CSTR$=CSTR$+Chr$(V+CODE)
  Next C
  Print CSTR$ : Rem remove if required
End Proc[CSTR$]
Procedure UNCODE[C$]
  Rem takes some coded text and converts it into normal English
  Rem The PASSWORD should be the same as the one used to encode the text
  PASSWORD$="123456"
  For C=1 To Len(C$)
  Rem Take each letter from the password one by one
  PC=C mod Len(PASSWORD$)+1
  CODE=Asc(Mid$(PASSWORD$,PC,1))-Asc("0")
  V=Asc(Mid$(C$,C,1))
  CSTR$=CSTR$+Chr$(V-CODE)
  Next C
  Print CSTR$ : Rem remove if required
End Proc[CSTR$]
```

4.5 Moving Between Rooms

4.5.1 A simple movement system

So far, I haven't included any mechanism to allow the player to move between the various rooms in the adventure. The first stage is to define our movement directions. Each direction can be stored as a single number, so North might be represented by direction number 1, South by 2 and so on. These numbers will be returned by the routine we'll be using to enter a movement command into the computer. I'll be explaining how this works in Sections 4.7 and 4.8.

For the time being, let's assume that the current direction number is automatically placed in a variable *D*. So if the user types GO NORTH from the keyboard, *D* will be magically loaded with the value 1. Our program can now carefully check whether there's an exit from the present location to the North. If it exists, the program can immediately jump to the routine we've created to handle the new location. This can be accomplished using a simple IF...THEN.

```
If D=1 Then Goto DINING_ROOM
```

DINING_ROOM is a label which indicates the starting point of the routine we've defined to

handle all the events which occur in the dining room. Remember that each location in our adventure will need its own separate subroutine in the program.

Normally, each direction will be assigned to a specific variable. This will allow us to define our movements using normal English.

```
If D=NORTH Then Goto DINING_ROOM
```

Here's a list of the standard movement directions.

Now for a more practical example of this technique. Have a look at the game map in Figure 4.3. We'll now define a series of movement instructions for the bridge. This has exits to the north, south, east, west and south-east. So we'll need five If..Then statements to handle the movement.

```
If D=EAST Then Goto REC_AREA
If D=WEST Then Goto AIRLOCK
If D=NORTH Then Goto OBS_DECK
If D=SOUTH Then Goto GANGWAY
If D=SE Then Goto QUARTERS
```

These instructions would normally be placed in the movement section of our separate BRIDGE routine. The advantages of this system should be fairly obvious. Because the movement routine is extremely simple, we can see the various possibilities at a glance. So it's extremely easy to change our adventure once we've created it.

As with the text descriptions, we can hide our movement commands safely away before we distribute our game. The direction variables can be replaced with the appropriate constants 1, 2, 3, 4 .. using the SEARCH/REPLACE option from the AMOS editor. We can also change the room labels to things like ROOM1, ROOM2 and so on.

We can even extend this system to change the available exits during the course of a game. This gives us a neat way of simulating locked doors or lifts. It's just a matter of adding an extra test to the If..THEN statement as follows

```
If D=EAST and F(HATCH)=READY
  Goto AIRLOCK
Else
  Print "The hatch is closed!"
Endif
```

Note that F is an array which will be used to hold a list of special variables known as flags. I'll explain these later.

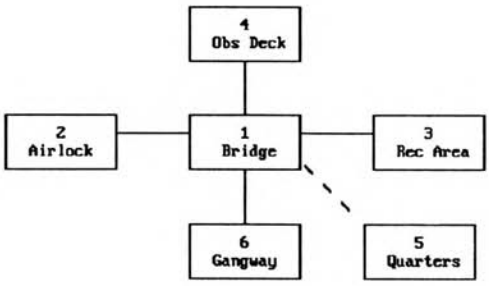

Figure 4.3: A typical game map

4.6 The Objects

4.6.1 Choosing the objects

The next step is to decide on the nature and position of our objects. These set up the tools which will be used by the player to solve the various puzzles in our adventure, and survive the cunning traps we are eagerly preparing for him.

It's important to realise that when I'm referring to an object, I'm actually talking about a vast range of possible entities. For the purposes of my discussion, an object can be treated as literally anything that can be manipulated by the player. Some objects can be picked up and examined, but others are much more intangible. Here are some typical common objects which can be found in an adventure:

Keys	Can be used to unlock doors.
Doors	May be opened
Holes	May be fallen into, or just looked in
Spells	Cannot be examined or picked up but may be cast
Buttons	Can be pressed
Red herrings	Intended simply to waste the player's time.

While we are choosing our objects, it's a good idea to jot down a few of the possible words which could be used to manipulate them. This will prove very helpful when we finally start generating the list of words required for the text analysis system (if we're using one). Each object in will be assigned its own unique identification number. When we are starting our adventure, it's best to assign these numbers to a list of standard constants. This allows us to refer to an object directly, without having to look up its number in a table.

For example:

```
CARROT=1:rem carrot is object 1
WATCH=2:rem watch is object 2
```

We can now test for the presence of the carrot using a line like:

```
If ITEM(1)=CARROT Then Print "What's up doc!"
```

4.6.2 Additional Information

In order to manage the objects in our adventure successfully, it's necessary to store a number of additional pieces of information.

The current location

This is simply the room number where the object can be currently found. It is normally held in a location array:

```
Dim LOC(1000):rem Where there are a maximum of 1000 objects.
```

Depending on the nature of our adventure, there may be several of these locations.

Uncreated

The object has yet to be discovered or created in the adventure. It is effectively in limbo until our program assigns it to a specific location.

Carried

The object is carried by the player.

Worn

This means that the object is currently worn by the character.

Part

The object is part of another object, so it cannot be manipulated directly.

These locations can be assigned values outside the allowable range of room numbers. Each special location would have its own particular number, which could be set up at the start of our program like this:

```
UNCREATED=10000:Rem denotes that the object hasn't been created yet
CARRY=20000:Rem represents an object which is carried by the player.
WORN=30000:Rem Indicates that the object is worn by the character
```

Depending on the nature of your adventure, there may well be other special locations as well. It's important to realise that not all objects in our adventure can be carried by the character. We therefore need to treat any immovable objects separately.

One possibility, is to define an array to store the weight of every object which appears in the adventure. Alternatively, we can use some sort of coding scheme to distinguish movable from fixed objects. This can be achieved by multiplying the location number in *LOC* by –1. So all fixed objects would be have a negative location value. Supposing object 2 represented a door at room number 10. The location value would be set to:

Room number * –1 = –10

We'll also need some way of storing in memory a description of each object. Since these descriptions are usually rather short, it's easiest to keep them in an normal array:

```
Dim OBJECT$(1000)
```

Finally, there is the ITEM array. This holds the object number associated with any particular word or screen icon. It is used to convert a raw word number into a reference to a specific object in the adventure.

4.6.3 Listing the objects in a particular room

Whenever we enter a new room, we'll need to list all the objects currently visible. Here's a small example routine:

```
LOOK:
DSP=0:Rem This is a simple value, used to trigger the initial message
Rem search for the location of each object
For O=1 To OCOUNT
  Rem The ABS bit converts the room number into a positive value
  Rem So we can display fixed objects as well as moveable ones
  Rem If object is at the current room
  If Abs(LOC(O))=R
    Rem If this is the first object, print out a neat header
    If DSP=0
      Rem STRING$(S$,N) generates a string holding N versions of S$
      Rem It produces a simple underline effect
      Print : Print "You see  " : Print String$("-",20)
      Inc DSP
    End If
    Rem print out the object description
    Print OBJECT$(O)
  End If
```

```
Next O
Rem print an extra line after the object text
If DSP<>0 Then Print
Return
```

This routine checks through each object in the adventure to see if it is positioned in the present room. If so, it displays its name on the screen.

4.6.4 The inventory

As the character moves through the adventure, any objects being carried will need to be moved accordingly. This can be accomplished by storing in an array the identification numbers of the required objects.

```
Dim INV(20)
```

The size of this array determines the maximum number of objects that can be carried by the player. The inventory can be printed out at any time using the following code:

```
INVENT:
Rem system command to display an inventory on the screen
APPARENT=0 : Print "You are carrying" : Print String$("-",16)
Rem If you're not carrying anything then print "nothing"
If HELD=0 Then Print "nothing" : Return
Rem Check through the inventory list in INV
For N=0 To 20
  O=INV(N) : Rem Get object number carried
  Rem Only print carried objects. Feel free to extend this
  Rem to WORN, or SPECIAL objects as well
  If LOC(O)=CARRIED Then Print OBJECT$(O) : APPARENT=1
Next N
Rem APPARENT holds the number of VISIBLE objects being carried
If APPARENT=0 Then Print "nothing"
Print
Return
```

4.7 Graphical User Interfaces

The core of any adventure program is provided by the procedures which input the user's commands and convert them into a form that can be directly understood by the computer. Originally all communication was handled using a text analysis system known as a parser. Nowadays, many adventures also include some sort of graphical interface. So commonplace commands such as Move, Get, Drop and Examine can be selected directly from the screen with the mouse.

Used properly, the graphical approach has a number of real advantages over its predecessor. Not only does it free the player from a great deal of unnecessary typing, but it's also far easier to operate. Important commands can be continually displayed on the screen, ready for instant use. This allows our players to concentrate on the problem solving aspects of the adventure, without having to worry about the exact phrasing of their commands.

The best thing about the graphical systems though, is that they are incredibly simple to produce. AMOS Basic contains an impressive range of instructions which practically automate the entire process. Creating a graphical interface is therefore just a matter of designing the various icons and positioning them on the screen.

4.7.1 Icons and buttons

All graphical user interfaces make use of a limited number of object types.

Icons are small rectangular images which can be clicked on with the mouse. The pictures are created with the aid of the AMOS Sprite definer, and are usually placed in their own separate Icon Bank (Bank 2). They can then be drawn on the screen using a simple PASTE ICON command.

Buttons are similar to Icons and can be selected from the screen in exactly the same way. Each button contains a small piece of text, enclosed by a rectangular box.

The text can be generated using any of the standard AMOS text commands. So providing we've enough memory, we can effortlessly create buttons in a variety of different fonts.

The box is usually produced with the AMOS Box command, although we can also exploit the Border$ function to surround our text with a standard window border. When a button is selected by the user, the text inside a button can be optionally adjusted to reflect the new situation. Suppose we had a button like that in Figure 4.4:

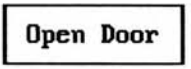

Figure 4.4: Open Door Icon

After the player had chosen this, the text could be changed as shown in Figure 4.5.

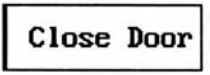

Figure 4.5: Close Door icon

Free text is a piece of text which is not enclosed by a border. A typical example of is the heading used to display the title of a menu.

All the above items can be freely mixed in our control panels as required.

4.7.2 Designing a user interface

When we are creating our control panels, it's vital that we use the correct type of object for each command. Icons should only be used if the associated action can be intuitively represented pictorially. If we can't think of a really obvious image, it's much better to define a button instead. Here are a couple of simple examples of the right and the wrong way of creating your objects.

Figure 4.6: Arrow icon

The Icon in Figure 4.6 should be familiar to any experienced graphic adventurer. It means Go Left, and will be reasonably obvious to most users.

Figure 4.7: A Jump icon

The icon in Figure 4.7 was supposed to mean Jump, but it could be interpreted in a number of different ways. It would be far better to replace it with a button as in Figure 4.8.

Figure 4.8: A Jump button

After we've designed our Icons and loaded them into the memory, we'll need to decide where they will be physically positioned onto the screen. In practice, it's well worth spending a bit of time over this, as it can make a great deal of difference to the playability of our adventure. Here are a few simple guidelines to help you along:

❑ **Keep it simple!**

If we attempt to pack too many Icons onto a single game screen, it will be impossible to use. Conversely, if there are too few, the layout will look totally ridiculous.

❑ **Group all Icons according to their function**

Take the trouble to ensure that all similar commands are positioned reasonably close together. An example of this can be seen on your Amiga's keyboard, which is split up into several groups of keys, each controlling a particular set of commands.

Also, think about the mouse movements required to accomplish common commands. Don't expect the player to continually drag the mouse back and forth across the screen to flick between simple functions such as movement and object manipulation. This will be a real irritation and will be quickly spotted by reviewers.

❑ **Place rarely used commands on their own separate control panel**

Allow the player to call up this panel using a single Icon on the main screen. This dramatically reduces the complexity of the system, without placing any undue limit on the number of available options. We might, for instance, decide to place all the system commands on their own screen. We could then assign the menu to an Icon with the shape of a disk or a SYSTEM button.

❑ **Add a check to dangerous commands such as QUIT or RESTART**

Only perform these functions after you've received confirmation by the user. This is incredibly easy to implement, and it adds a valuable safeguard to our game.

```
ADV_QUIT: Input "Are you sure [Y,N]";C$:C$=UPPER$(C$)
  If C$="Y" Then End
  Return
```

4.7.3 Creating a graphical user interface

So much for the theory. I'll now go on explain how we can actually create one of these interfaces in our own programs. As I've mentioned previously, the first step is to draw our

Icons using the AMOS Sprite Editor, and convert them into an appropriate .abk file. We can now enter them into memory using a line such as:

```
Load "Iconfile.abk",2
```

We can then generate the various zones which will be used to enter the user's commands. AMOS Basic allows us to divide the screen into a number of invisible rectangular screen areas. These can be placed around each Icon or Button using the following instructions:

RESET ZONE

Initialises all the AMOS zone routines. Call it at the start of your program.

RESERVE ZONE n

Reserves enough memory for *n* zones

SET ZONE z,tx,ty TO bx,by

Defines zone number *z* at screen coordinates *tx,ty* to *bx,by*.

z=MOUSE ZONE

Gets the number of the first zone underneath the mouse pointer. If the pointer is not over a zone a value of zero will be returned.

The general procedure is:

- First make a count of the maximum number of Icons and Buttons which will appear on the screen at any time. We can now reserve memory for our screen zones using the RESERVE ZONE command:

```
Reserve Zone 100:rem allocate space for a hundred zones
```

This command should be performed at the start of our adventure.

- Initialise all the zones in the current control panel using RESET ZONE. Note: If we are intending to create a number of separate control panels, we'll need to execute this command whenever we redraw our icons on the screen. However, this will only be necessary if the size and positions of the zones need to be changed. Providing we draw our new Icons directly over the originals, we can happily reuse the same zones again and again.

- Successively draw each Icon and Button on the screen, enclosing it with a screen zone. This can be accomplished with the SET ZONE command:

```
Procedure MAKE_ICON[IM,ZN,X,Y]
  Rem IM is the number of an icon from the icon bank
  Rem ZN is the number of the zone we wish to define
  Rem X,Y are the new screen coordinates of the icon zone
  Paste Icon X,Y,IM
  Rem Get size of Icon from Bank using a bit of sneaky peeking
  A=Icon Base(IM)
  W=Deek(A)*16 : H=Deek(A+2)
  Rem Create a zone around the icon
  Set Zone ZN,X,Y To X+W,Y+H
  Rem Draw a box around the icon as well (optional)
  Rem Box X,Y To X+W,Y+H
End Proc

Procedure MAKE_BUTTON[T$,ZN,X,Y]
 Rem Makes a button using the text in T$
```

```
 Rem ZN=The zone number to be set up
 Rem X and Y are the screen coordinates of the top left of the button
 Rem You could use PRINT with BORDER$
 Rem but graphic text works better
 W=Text Length(T$) : H=8 : Rem Assumes 8x8 text font
 Rem Draw a box around the button
 Box X,Y To X+W+2,Y+H+2
 Rem Position the button inside this box
 Rem TEXT takes the Y coordinate from the bottom of each character
 Rem A correction factor is available from the TEXT BASE function
 B=Text Base
 Rem Draw the text on the screen
 Text X+2,Y+B+2,T$
 Rem There's also a ZONE$ command for use with PRINT
 Rem but this sets the zone around just the text, ignoring any border!
 Rem Define the zone
 Set Zone ZN,X,Y To X+W+2,Y+H+2
End Proc
```

And that's it! We simply check the zones at start of our room routine, and load the number of the current option into ACTION. We can now add an explicit test for these options into the routines used to handle our Local or Global commands. Any additional objects which need to be manipulated by the command can be selected using a separate screen or may be entered directly from the keyboard. Similarly, directions can be handled in exactly the same way. Define the first eight screen Zones for the arrow Icons used for the movement commands. If the ACTION number is less than or equal to 8, the player has obviously entered a direction and the character should be moved appropriately.

Examples of these routines in action can be found on the reader convenience disk, and in the character generator in Chapter 5. Here's how you'd use these routines in a ROOM definition.

```
ROOM1: Rem Don't type this in. It's just a demonstration.
R=1
Gosub HI:rem do High priority events
rem Handle description
: :
Gosub CONTROLS:rem enter users commands from the screen
If ACTION<=8 Then D=ACTION else ACTION=ACTION-8
If D<>0
  Rem Player entered a direction
  If D=EAST: Goto DINING_ROOM: Endif
  If D=WEST: Goto KITCHEN : Endif
  If D=NORTH : Goto LIBRARY : Endif
else
  Rem Player entered a command
  CHECK_GLOBAL : rem check for a global command like wear
  Gosub CHECK_SYSTEM : Rem check for system commands
  Rem First control panel
  If ACTION<>0 and PANEL=1
  If ACTION=FGHT: FIGHT : Endif
  If ACTION=RETRT: RETREAT :Endif
  Endif
  rem handle second control panel (used for Magic?)
  If ACTION<>0 And PANEL=2
  If ACTION=CAST : CAST_SPELL :Endif
  If ACTION=ZAP : CAST_ZAP :Endif
  If ACTION=X : PANEL=1 :Endif
  Endif
Endif
```

```
Goto ROOM1
: :
Rem Example of secondary control panel
If ACTION=MAGIC
  Rem calls a separate procedure to create the new panel
  PANEL=2
  CREATE_PANEL
Endif
```

4.8 Understanding Text

You may be wondering at this point why we need a separate routine to understand the user's text. After all, it's perfectly feasible to input the user's commands directly into a string variable with Input. We could then test this variable using a successive set of IF..THEN statements.

```
Input C$
rem get_rock procedure defined separately
If C$="get rock" Then Gosub GET_ROCK
If C$="kill dragon" Then gosub KILL_DRAGON
```

The flaw with this simple approach is that it ignores the sheer richness of the English language. A single command such as *Kill dragon* could also be expressed as:

Kill monster

Attack dragon

Destroy dragon

If we were to limit our program to simple comparisons between the possible input values, we would be forced to check each of the alternatives individually. Otherwise, it could take the player hours of effort to come across the exact phrasing our game required. In practice, we'd need thousands of IF..THEN statements to implement even the smallest adventure. This is obviously crazy.

4.8.1 The verb noun parser

The solution is to allow each command to be entered using a number of common synonyms. Synonyms are just lists of words with the same or similar meanings. So *destroy* and *attack* are synonyms of the word *kill*. A special dictionary of these synonyms, called a Thesaurus, lists all the possible alternatives for most English words or phrases, and should be available from your local library. Here's a typical definition and some of its synonyms:

Typical
Average
Characteristic
Classic
Essential
Normal
Model

The job of the text parser is to convert a complex English sentence into a form which can be unambiguously recognised by the computer. This will involve converting any common synonyms of a word into a single unique identification number. We will now be able to perform just one test for all combinations of a certain command with just a single IF..THEN instruction:

```
If ACTION=KILL and ITEM=DRAGON Then Gosub KILL_MONSTER
```

The first requirement is to split up a sentence into its individual words. In order to do this, we'll need to be able to manipulate character strings freely and easily. So I'll now embark on a crash course for any beginners out there! If you're already familiar with INSTR$, MID$ and LEFT$, you should be able to skim through this fairly quickly. You can even jump straight to Example 4.3.

Since most English words are separated by spaces, we can isolate all our commands using the AMOS Basic INSTR function. INSTR (pronounced instring) is a simple function which allows us to search a string variable for any specific group of characters. The basic form is just:

p=INSTR(text$,search$)

This checks *text$* for the first occurrence of the characters in *search$*. After the search is complete, INSTR will return either the position of the character or a value of zero. Note that all positions are measured from left to right, starting at 1. So the available position numbers range from 1 to the length of the string. That is:

String "abcdefgh"

Position 12345678

Here are a few examples:

```
print instr("abcdefg","a")
```

Returns the position of the *a*, which is 1.

```
print instr("abcdefg","d")
```

Finds the *d* at position 4.

```
print instr("abcdefg","z")
```

Returns a zero as there's no z in the search string.

The clever thing about INSTR is that it can be used to find the spaces separating the words in our sentence:

```
print instr("the game makers manual"," ")
```

This returns the position of the first space in the text, and generates a value of 4. It finds the space just after *the*. That's fine for the first space, but how do we check for all the other spaces in the text, such as the one between *game* and *makers*. Well, as you might expect, there's a second version of INSTR which allows us to move the starting point of the search anywhere with our text.

=INSTR(text$,search$,p)

p is now the position of the first character to be checked in *text$*. If *p* is less than 1, or greater than the length of *text$*, we'll get an error.

Examples:

```
print instr("the game makers manual"," ",5)
```

produces a result of 9. See how I've carefully positioned the search just after the initial space.

```
"the game makers manual"

 123456789...
```

What starting point will we need to find the final space in the string? Can you work it out?

```
print instr("the game makers manual"," ",10)
```

displays the position of the space between *makers* and *manual* (16).

Once we've found the beginning of our words, we'll need to load them into a series of individual string variables. With the help of the MID$ command, we can easily grab the sections of our text between the spaces.

S$=MID$(text$,p,n)

MID$ (pronounced midstring) returns the first *n* characters in *text$*, starting a position *p*.

Here are some examples for you to play around with:

```
print mid$("abcdefgh",3,3)
```

returns *cde*.

```
print mid$("the game makers manual",1,3)
```

generates *the*.

```
print mid$("the game makers manual",5,4)
```

game

```
print mid$("the game makers manual",10,6)
```

makers

```
print mid$("the game makers manual",17,6)
```

manual

If you examine this sequence carefully, you should notice a pattern. Remember, the spaces found by our previous INSTR commands were at 4, 9 and 16. The first MID$ has started at 1, and grabbed the characters up to the initial space.

Suppose we stored the search position in the variable *POS*, and the location of our space in *TEST*. We'd begin by loading POS with 1 and finding a space with:

```
  TEST=Instr("the game makers manual"," ",POS)
```

After we'd called this function, *TEST* would be loaded with a value of 4. We could then use a MID$ function to load up our word into a string.

```
print mid$("the game makers manual",1,3)
```

Now, the 1 is held in *POS* and the 3 is equal to:

```
TEST-POS = 4-1 = 3
```

So our final command would be:

```
Print Mid$("the game makers manual,POS,TEST-POS)
```

We could now move *POS* to the position of the next word using:

```
POS=TEST+1
```

where *TEST+1* is the first letter after the space.

If we repeat this process, we'll list all the words in the current message. Except, of course, for the last one! That's because the final word doesn't end in a space. So *TEST* will be loaded with zero, and the system will collapse. We can however, check for this in our program, and work out the number of characters in the final word.

In our example, we displayed this word with:

```
print mid$("the game makers manual",17,6)
```

17 is the position of *m* in *manual*. It's equal to *POS*.

6 is the number of letters in our word. And the length of the *game makers manual* bit is 22. What's the relationship between these numbers? If you think about it you'll quickly recognise that the required value is given by the following equation:

Length of word=Length of string – position of first letter in word +1

6 = 22 – 17 + 1

The length of our string is returned directly by the LEN function from AMOS Basic:

```
Print Len("the game makers manual)
```

22

So in order to print out the final word, we will need to use the following instructions:

```
L=Len("the game makers manual")-POS+1
Print Mid$("the game makers manual",POS+1,L)
```

Hopefully, you should now be a perfect position to understand the SPLIT_WORD routine in Example 4.3. The only new stuff will be the LEFT$, UPPER$ and SPACE$ commands used to convert our words into a standard format.

l$=LEFT$(s$,n)

Grabs the first *p* characters from *S$*, reading from left to right.

```
Print Left$("123456789",1)
1

Print Left$("123456789,5)
123456

Print Left$("the game makers manual",0)
the game
```

In Example 4.3, LEFT$ is used to chop any excess characters from our words, and ensure that they are all a maximum of seven letters long.

u$=UPPER$(l$)

Just converts any letters in *l$* into capitals.

```
print UPPER$("the game makers manual")
```

becomes:

THE GAME MAKERS MANUAL

Again, this ensures that all the user's commands will be in one simple format.

S$=SPACE$(n)

Loads *s$* will a string of exactly *n* space characters. It allows us add extra spaces to our words, ensuring that they are all exactly seven characters long. Anyway, without any no more ado, here's the word splitter!

Example 4.3: Separating words

```
Rem routine to enter a line of text and split it into single words
SPLIT_WORDS:
Rem holds input text
COUNT=0 : POS=1 : Rem Initialise word count and current search position
Rem Main loop
Repeat
  Rem Find the next space
  TEST=Instr(T$," ",POS)
  Rem if space is found
  If TEST<>0
    Rem get number of characters in the word
    L=TEST-POS
  Else
    Rem Get number of characters in last word
    L=Len(T$)-POS+1
  End If
  Rem convert word into upper case
  W$=Upper$(Mid$(T$,POS,L))
  POS=TEST+1 : Rem advance search position
  Rem Ignore THE,AN, or A before a word
  Rem You could extend this to ignore words like:
  Rem TO, IS, AT etc
  If(W$<>"THE") and (W$<>"A") and (W$<>"AN")
  Rem If word is greater than seven characters long
    Rem LEN(W$) returns the number of characters in the string W$
    If Len(W$)>=7
      Rem load the first seven characters into WORD$(COUNT)
      WORD$(COUNT)=Left$(W$,7)
    End If
    Rem If word is less than seven characters long
    If Len(W$)<7


      Rem Pad the end of the word with blank spaces
      Rem Space$(N) returns a string of N spaces
      WORD$(COUNT)=W$+Space$(7-Len(W$))
    End If
     Inc COUNT : Rem increase word count
  End If
Until TEST=0 or COUNT=11
Rem Assumes that WORD$ is a maximum of 10 elements in length
Return
```

WORD$ is an array which will be used to hold our words. This will need to be dimensioned at the start of our adventure.

Here's a small program which tests this routine in action.

```
Dim WORD$(10)
Do
  Line input "->",T$
  Gosub SPLIT_WORDS
  For W=0 to COUNT-1 : Print WORD$(W) : next W
Loop
```

Don't forget to tag the SPLIT_WORDS routine to the end, before trying to run this program. Otherwise you'll get an error message.

LINE INPUT incidentally, is just an advanced version of INPUT. The only difference between the two instructions, is that LINE INPUT separates each variable with Return rather than a comma. Ok?

Once we've separated out the individual words in our sentence, our program can finally begin to interpret them. The simplest parsers use the verb-noun system. This allows the user to enter sentences of up to two words in length.

The verb specifies an action to be taken while the noun indicates the object to be manipulated. A valid sentence can begin with either a verb or a direction. So the following phrases are all legal in this system:

run

east

get carrot

Since it's completely ridiculous to start off a sentence with a noun, this needs to be automatically rejected by the parser. Therefore a command such as:

carrot

should generate a response from our adventure like:

I'm sorry, but you are not allowed to "CARROT" something.

The action of a verb-noun parser can be summarised by the pseudo-code in Figure 4.9.

Note: You are strongly recommended to keep the computer's responses to improper English as polite as possible. Don't fall into the trap of using phrases which are sarcastic or rude. No matter how funny or clever these seem at the time, they will quickly get under the player's skin. After the 14th instalment of:

Learn to type DUMMY!

even the most patient player is likely to feel justly aggrieved!

I've already demonstrated how a phrase can be split into words in Example 4.3 The parser will also need the following routines:

check if word is a noun
check if word is a verb
check if word is a direction
perform action

Figure 4.9: Pseudo-code expansion for a verb-noun parser

```
split sentence into words
if number of words>2
  then
    output "I'm not sure what you mean"
  endif
if word one is a direction
  then
    move character
  endif
if word one is a verb
   then
     if number of words=1
     then
       Perform action
     else
     If second word is noun or direction
     then
       perform action on object
     else
       Print "What ";word$(2);"?";
     endif
     endif
     else
  print "you can't ";word one;" something"
endif
```

AMOS Basic contains a number of powerful search commands which make it very easy for our program to check through a list of synonyms. We can, for instance, use the INSTR instruction to test the words against of range of possibilities held in a string variable. Each type of word will have its own individual search string. This will contain a complete list of all words in the particular category we wish to check.

It's common practice to restrict each response to just the first few letters of each word. This simplifies the search procedure enormously. The number of significant letters we use for each word will depend on the amount of memory we are prepared to allocate for the search string.

A word size of seven is ideal for this purpose, as it combines a reasonable measure of accuracy with only a minimal memory overhead. Each list can be loaded directly into the string using a simple assignment statement:

```
VERB_LIST$="GET    PUT    ATTACK "
```

After we've generated a list, we can search for our words using the INSTR statement:

```
V=Instr(VERB_LIST$,WORD$(1)):ACTION=V/7:rem get verb number
```

The workings of this system can be seen from Figure 4.10.

```
String   : "GET   PUT   ATTACK "
Position:  1     8     15
ACTION   :  0     1     2
```

Figure 4.10: Converting a word into a number

Notice how V/7 has been rounded down to the nearest whole number. Although 15/7 is actually 2.14, AMOS automatically throws away the decimal bit, and leaves us with just 2. Unfortunately, the above instruction produces a different number for every single word. What

our program needs is some way of grouping together all words with the same meaning. The solution is to store a separate identification number for each possible synonym in an array.

```
Dim VERB_NO(100)
```

The typical contents of the VERB_NO array can be seen in Figure 4.11.

Element number	Verb number	Verb
1	1	GET
2	2	DROP
3	2	PUT
4	3	KILL
5	3	ATTACK
6	4	RETREAT

Figure 4.11: Verb numbers

Elements two and three of this array both have exactly the same *verb_no* number. They therefore represent a single command from the player. Similarly, KILL and ATTACK have a *verb_no* of 3, and will therefore be treated as synonyms by our program. The arrays might be loaded from data statements using the following code:

```
Data 6:rem number of words in list
Data "GET",1,"DROP",2,"PUT",2,"ATTACK",3,"KILL",3,"RETREAT",4
```

This list could be loaded into *VERB_LIST$* at the start of your program.

```
Read V
For VN=1 To V
Read V$,SYN
VERB_NO(VN)=SYN
  rem Pad word with spaces
  L=Len(V$)
  If L<7 Then VERB_LIST$=VERB_LIST$+V$+Space$(7-L)
  rem Only use the first 7 letters of word
  If L>7 Then VERB_LIST$=VERB_LIST$+Left$(V$,7)
  If L=7 Then VERB_LIST$=VERB_LIST$+V$
Next VN
```

The same system could also be used for our nouns. We could load up the synonyms used to describe an object into an NOUN_NO array. The format of this array will be identical to VERB_NO. Here's a complete analysis system for you to examine.

```
Rem Simple verb noun parser
ENTER_WORDS:
Rem Enter a line of text
Line Input "->";T$
Rem Split text into its individual words
Gosub SPLIT_WORDS
Rem Initialise variables
D=0 : ACTION=0 : ITEM=0
Rem Check number of words
Rem Note Return leaves the procedure immediately
If COUNT>2
  Print "Err..I'm not sure what you mean"; : ER=1 : Return
```

```
Endif
Rem First word
Rem search for an exit
D=Instr(A_DIR$,WORD$(0))
If D<>0 Then D=A_DIR_NO(D/7) : Return : Rem Return a direction
Rem if word is not an exit, then try for a verb
V=Instr(VERB$,WORD$(0))
Rem if first word is not a verb or direction
If V=0
  Print "I don't know how to ";WORD$(0);" something" : ER=1
  Return : Rem Exit with an error
Else
  ACTION=VERB_NO(V/7) : Rem get verb number
  Rem Is verb=GO? If so, test second word
  If V=1
   Rem find exit
    D=Instr(A_DIR$,WORD$(1))
    Rem Get direction number and leave
    If D<>0 : D=A_DIR_NO(D/7) : Return : End If
  End If
  Rem look for second word in the noun list
  N=Instr(NOUN$,WORD$(1))
  If N=0 : Print "What ";WORD$(1);"?" : ER=1 : Return : End If
  Rem if noun is found then get object number from noun_no array
  ITEM=NOUN_NO(N/7)
End If
Return
```

This routine does everything needed to enter and interpret our commands. It reads a line of text from the keyboard with LINE INPUT, and then calls the SPLIT_WORDS routine to load a list of words to be held in the WORD$ array. I've also introduced a separate mechanism for checking whether a word is a direction. This uses the A_DIR$ and A_DIR_NO arrays. The first is just a list of the possible directions:

```
A_DIR$="NORTH  N      SOUTH  S      "
```

The second holds the direction number for each word in this string. So NORTH and N both have exactly the same direction number as they represent a single movement for the player. Before we can test ENTER_WORDS in anger, we need to define the arrays NOUN$, VERB$ using line like:

```
NOUN$="CARROT DOG    ORANGE HAT    ": VERB$="GET    DROP   KILL   "
```

See how I've spaced out these words so they are always seven characters apart.

4.8.2 Expanding the parser

Although verb-noun parsers are undoubtedly useful, they are not exactly state of the art. Most text adventures now incorporate far more advanced text recognition systems, which can easily interpret a wide range of common English expressions. I'll now examine a number of simple techniques which can be used to enhance the standard verb-noun system considerably.

Prepositions

A preposition is a word which is used to specify either the location of a noun or the means used to manipulate it. Common English prepositions include *inside, under, with, by, at.* The ability to recognise phrases containing prepositions increases the apparent intelligence of a parser dramatically. All the following phases could now be interpreted successfully by the computer.

```
get gun from pocket

look around corner

kill monster using axe

open safe with key
```

The typical phrase now includes one verb, two nouns and a preposition. The first noun in this phrase is known as the subject and the second as the object. Since a pseudo-code description of the analysis system looks insanely complicated, I'll limit myself to a practical demonstration of the routine in action. Don't worry if it's not immediately obvious how it works, as you can use it directly in your adventures, without having to understand a single line:

```
XPARSER:
Rem verb noun preposition parser
Rem set up initial values
Rem ER is an ERROR flag. It's set to one if there are any problems
Rem D holds the direction number
Rem ITEM(1) and ITEM(1) the numbers of any nouns
Rem PREP holds the preposition number
Rem And NUM the numerical value of ITEM(1)
Rem If ITEM(1) is not a number, NUM will be zero
ER=0 : D=0 : ITEM(1)=0 : ITEM(2)=0 : ACTION=0 : PREP=0 : NUM=0
Rem This holds the word number of the verb "GO". Set as appropriate
GO=1
Line Input "->";T$ : Rem Enter a line of text
Rem Check for an empty line
If T$="" Then Return
Gosub SPLIT_WORDS : Rem Split text into its individual words
D=Instr(A_DIR$,WORD$(0)) : Rem search for an exit
If D<>0 Then D=A_DIR_NO(D/7) : Return : Rem Return a direction
Rem if word is not an exit, then try for a verb
V=Instr(VERB$,WORD$(0))
Rem if first word is not a verb or direction
If V=0
  Print "I don't know how to ";WORD$(0);" something" : Print : ER=1
  Return : Rem Exit with an error
End If
ACTION=VERB_NO(V/7) : Rem get verb number
Rem This next bit's for the LOAD and SAVE commands I'll be showing
Rem you later. It allows us to type LOAD or SAVE filenames
Rem These filenames can be up to seven letters long
Rem and should not include the drive name. ADV will be added to the end
If ACTION=ADVLOAD or ACTION=ADVSAVE Then Return
Rem Is verb=GO? If so, test second word
If V=GO
  Rem find exit
  D=Instr(A_DIR$,WORD$(1))
  Rem Get direction number and leave
  If D<>0 : D=A_DIR_NO(D/7)
    If COUNT>2 : Goto THREE_WORDS : End If
    Return
  End If
End If
Rem VAL converts a string of characters such as 1111 to the equivalent no
Rem If WORD$(1) holds something silly like AAAAA, we get a zero
NUM=Val(WORD$(1)) : If NUM>0 Then Return
Rem If sentence contains two words check if second word is a noun
```

```
If COUNT=2
  W=1 : N=1 : Gosub FIND_NOUN
  Return : Rem Leave parser
End If
Rem Check word 2 for a preposition and then check word three for noun
THREE_WORDS:
If COUNT=3
  W=1 : Gosub FIND_PREP : W=2 : N=1 : Gosub FIND_NOUN
  Return : Rem Exit
End If
Rem sentence contains four words
If COUNT=4
  If D=0 : W=1 : N=1 : Gosub FIND_NOUN : End If : Rem is word two a noun?
  W=2 : Gosub FIND_PREP : Rem is word 3 a preposition?
  NUM=Val(WORD$(3)) : If NUM<>0 : Return : End If
  W=3 : N=2 : Gosub FIND_NOUN : Rem is word four another noun?
End If
Return
FIND_NOUN:
Rem Check word W to see if it refers to item number N
OB=Instr(NOUN$,WORD$(W))
Rem Word W is not a noun
If OB=0 Then Print "What ";WORD$(W);"?" : Print : ER=1 : Return
ITEM(N)=NOUN_NO(OB/7)
Return
FIND_PREP:
P=Instr(PREP$,WORD$(W)) : Rem CHECK For A PREPOSITION
If P=0
  Print "I don't understand ";WORD$(W);" in this context"
  Print : ER=1 : Return
End If
PREP=PREP_NO(P/7) : Rem if preposition has been found then get its number
Return
```

This parser should prove more than adequate for the vast majority of our adventures. It's certainly preferable to a verb-noun system. But if you enjoy experimenting, there are many further improvements you could make. We might decide to extend it to understand sentences involving more than two nouns, for example:

attack monster inside room with axe

We could also include the ability to interpret lists, adverbs, adjectives and questions. Adjectives and questions are easy, as the structure of the commands are very similar to the NVP system. There's no logical difference between instructions such as:

look inside shoe

and

get green card

or

who is amos

All we do, is define WHO as a verb, and create extra pronouns such as IS and GREEN. An example of this technique can be seen in the demo game at the end of this chapter.

Adverbs

Adverbs describe the way we do a verb. They're usually words ending in *ly* like *slowly, aggressively* and *stealthily*. In this case we'll need to create an extra word list, and add some extra rules to the parser. It would need to handle commands such as:

```
run away quickly
```

```
gently lift the scroll
```

Each different category would require its own individual search string:

```
ADVERB$="QUICKLYSLOWLY GENTLY "
```

We'd also need to keep track of the synonyms using an array of word numbers:

```
A=Instr(ADVERB$,WORD$(W))
rem ADVERB$ holds a list of the synonym numbers of each word
if A<>0 then ADVB=ADVERB(A/7)
```

Lists can be separated by commas and might be entered into the computer using a line such as:

```
get apple,pear,banana,orang-utan
```

The word splitter procedure would automatically load the entire list into word number 2. We could now add a test for this list into the FIND_NOUN routine.

As you can see, it's easy to get carried away, and to concentrate all our effort into the creation of ever more powerful text recognition systems. This is probably a mistake! Although software reviewers seem to love the advanced parsers, many users quickly fall back to the original noun verb system. The noun verb system may be old-fashioned, but it's superbly direct. Seasoned adventurers quickly realise that the chances of an adventure misinterpreting their commands is reduced enormously if they keep their instructions as straightforward as possible.

Despite having played some of the latest adventures on the market, I've yet to encounter a parser which could understand absolutely everything I threw at it. So there's nothing to be ashamed of if you decide to restrict yourself to just the simplest text parser in your adventures. This will allow you to devote most of your energy into creating a really interesting scenario.

If your adventure captures the player's imagination, and includes absorbing and challenging puzzles, the quality of the parser is frankly irrelevant. Having said that, I've had great fun extending and creating the NVP parser! The choice is up to you.

4.9 Handling the Events

4.9.1 Flags

All the events in our adventure will be controlled through a set of variables known as flags. These store all the various pieces of information needed to determine the outcome of an event or the fulfilment of a condition. Here are a few examples of how these flags might be used in practice:

SAFE_OPEN Set to 1 if the safe was open, otherwise 0.

ALIVE Set to 0 if the player is killed. Normally 1.

TORCH Stores the number of turns the current torch will last.

HUNGER Set when the player is hungry.

It's theoretically possible to store these flags using normal Basic variables. Unfortunately we'd now need to save each variable by name during the Save game option. This would be unbelievably tedious, and it would be easy to miss out an essential variable from the list during development. It is much safer to hold the main variables in a single array. We could define this at the start of our program using:

```
Dim F(1000)
```

We could read to each flag from the array using the index number. So the FOOD flag might correspond to F(1), the TORCH flag to F(2) and so on. This may be easy for the computer to understand, but it's a pain in the neck for mere humans. We can however, simplify the system by assigning all the numbers to a set of Basic variables at the start of our program. We can now refer to each flag using an appropriate name:

```
F(TORCH)=10:rem torch=1
```

Each variable would need to be defined separately during the initialisation process:

```
TORCH=1:SAFE_OPEN=2:ALIVE=3:HUNGER=4
```

After we'd created our game, we could later replace these variables using the AMOS CHANGE command. This would stop an unscrupulous player searching through the program text for hints. (Warning! Only use a COPY of your game for this purpose!. You may need to edit it in the future!)

4.9.2 Generating the events

All the events which can happen to the adventurer can be divided into three different categories:

High priority

Performed at the start of the room routine, before the player gets a chance to enter any commands.

Low priority

Executed at the end of the room routine.

Local

Are local to a specific room.

Here is an example of how a high priority event could be written in AMOS Basic:

```
DEATH:
If F(ALIVE)=1 then Return:rem Players character is alive and well
Rem If character dies, jump out of routine
Rem POP allows us to skip the RETURN and jump out of GOSUB with GOTO
Pop:Goto AGAIN:rem Another Game?
return
```

The "alive" flag might be set by a low priority event like this:

```
HUNGER:
Dec F(FOOD)
If F(FOOD)<0 Then Print "I'm Hungry":V(HUNGER)=0
If F(FOOD)<-5 Then Print "I'm starving"
If F(FOOD)<-10
  Print "If I don't eat something soon I'll probably die!"
Endif
F(FOOD)<-15 Then "I've starved to DEATH!":F(ALIVE)=0
Return
```

In order keep our programs nice and compact, it's sensible to load all our events and commands into just two large routines. Here are some examples:

```
HI:
Rem Handles hi-priority events
Rem This bit just temporarily sets the SHORT variable to 1
Rem after the user enters EXAMINE or LOOK
Rem It forces a long description of the current surroundings
Rem See later
If(ACTION=DESC or ACTION=EXAMINE) and COUNT=1 Then U=1 : LOOK=1 Else
LOOK=0
Rem Chooses between a short and long description
If(VISIT(R)>0 and VERBOSE=0 and LOOK=0) Then SHORT=1 Else SHORT=0
Inc VISIT(R)
Return

LO: Rem Low priority events
Rem Check if the characters fit enough to move
If F(HEALTHY)=0 and D<>0
  Print "You'll need to stand up first"
  Print : Return
End If
Rem Handle errors in the users commands
If D<>0 Then Print "You can't go in that direction" : Print : Return
If U=0 and ER=0 Then Print "I'm sorry, I don't understand" : Print :
Print : Return
U=0
Return
```

Remember that the essential difference between a high and a low priority event is the order they are executed in the program. Otherwise, the programming techniques required to generate them are exactly the same. Both events should be called directly in our program at the start of each room routine.

In the middle of the room routine, we can place all the local events that can occur in the current location. Here's a typical room routine:

```
SOUTH_CORRIDOR:
R=2 : Gosub HI
If SHORT
  Print "South Corridor "
Else
  Print "You are on a long featureless corridor stretching to the north"
  Print "To the west there's a door with a";Pen$(4);
  Print " RED ";Pen$(2);"sign on it"
End If
Gosub USER:Rem Call up Parser and interpret system and global commands
Rem Now for our local commands
If D=WEST Then Goto CELL
If D=NORTH Then Goto NORTH_CORRIDOR
If ACTION=AREAD and ITEM(1)=SIGN
  U=1
  Pen 4 : Print : Locate 1,
  Print Border$(" DANGEROUS ANIMALS",1) : Print : Print : Print
  Pen 2
End If
Gosub LO
Goto SOUTH_CORRIDOR
```

4.10 Acting on the User's Commands

In order for an adventure to work, there has to be some way of entering the player's commands, and choosing the relevant action. I've already shown you how you can convert these instructions into a numerical format with the help of a text parser or graphic handler. We'll now step back a little, and demonstrate how these routines can be used in one of your own games.

After we've analysed the text using a parser or read the icons from the screen, our program will have the following information.

ACTION

Contains the number of the command which has been input by the user.

ITEM(1)

Holds the number of the object to be affected by the command in ACTION.

ITEM(2)

Holds the number of an item which will be used to manipulate the previous object. Set to 0 if it doesn't exist.

PREP

Loaded with the Preposition number. This is only really useful in a text adventure.

D

Direction number. Note that this will be automatically set to zero if the user hasn't entered a direction.

COUNT

This will contain the number of words entered by the user.

4.10.1 Interpreting commands

In a graphics adventure, interpreting our commands is easy, as there's usually only one action or direction for each icon. But in a text adventure, we need to check carefully for each specific combination of words. The parser helps in this, by converting each word into a single number, and loading these values into the appropriate variables. However, there's still quite a bit of work ahead for our program. Here's a typical example:

```
If ACTION=WEAR and ITEM(1)=SHOE and LOC(SHOE)=CARRIED
  U=1
  Print "Worn"
  LOC(SHOE)=WORN
End If
```

This checks for the command *WEAR SHOE.*

WEAR and *SHOE* are simple variables containing the item numbers and action numbers associated with each word. These were defined during my program's initialisation phase. See how I've tested whether the shoe is carried with:

```
LOC(SHOE)=CARRIED
```

The line `U=1` is as part of the error handling routine. It tells our program that the current

combination of words has been successfully recognised. If not, our LOW priority routine can display an appropriate error message.

```
LO:
Rem Check for a wrong direction
If D<>0 Then Print "You can't go in that direction" : Print : Return
Rem Check for an unrecognised word combination
Rem ER is set to one if the parser has already detected an error
If U=0 and ER=0 Then Print "I'm sorry, I don't understand" : Print :
Print : Return
U=0
Return
```

More examples of this system can be found in the Demo game at the end of this section.

4.10.2 Global commands

Global commands may be entered at any point in the adventure. Each can be implemented using a single AMOS Basic subroutine. The PICK, DROP and INVENT commands are typical.

```
rem Text versions
Rem OB holds the number of the object we wish to pick up
rem LOC() stores the location of each object
rem OBJECT$() contains text used to describe each object
Rem INV() is array used to hold the list of objects carried by player
rem CARRIED,WORN hold constants which indicate the status of an object
rem R= Number of current room. HELD=No of objects held by player
PICK_UP:
Rem get the object number we wish to get
OB=ITEM(1)
Rem Can we pick it up?
If LOC(OB)<0 and Abs(LOC(OB))=R
  Rem Nope!
  Print "You can't pick up ";OBJECT$(OB) : Return
End If
Rem Is it already carried or worn?
If LOC(OB)=CARRIED
  Print "But you are already carrying ";OBJECT$(OB) : Print : Return
End If
If LOC(OB)=WORN
  Print "You'll need to remove it first" : Print : Return
End If
Rem Or isn't it here?
If LOC(OB)<>R
  Print "It isn't here" : Print : Return
End If
Rem Now get our object
Inc HELD : Rem increase number of items carried
If HELD>20 Then Print "It's too heavy" : Return
Rem Get object
Gosub GRAB
Print "You picked up ";OBJECT$(OB) : Print
Return
GRAB:
Rem Updates our inventory, and sorts it!
INV(HELD)=OB : LOC(OB)=CARRIED : Sort INV(0) : Return
```

Note that since the order of the objects in unimportant, I can search through the list at high speed using the AMOS Basic MATCH command.

f=MATCH(ARRAY(0),S)

Takes a sorted array, and searches through it for item *s*. If the item is successfully found, the element number is returned in *f*. Otherwise MATCH will return a negative value. The ARRAY(0) bit is required by the MATCH command by the way. It's just the name of the first element in the array.

You may be wondering at this point why I've used INSTR$ rather than MATCH in my text analysis system. The answer's simple! We'd need to keep a list of all our words in ascending order! Every time we added a new word to our list, all the identification numbers of our words would change completely! Yugh!

Before using MATCH we need to sort our INV array using SORT

SORT(ARRAY(0))

This shuffles the elements of the array around, so that their contents are in ascending order!

If you examine this example carefully, you will probably be wondering why I bothered to sort the INV array. This is necessary to allow you to search through it using the MATCH command. Anyway,back to our adventure!

Here's the DROP and INVENT to complete the set:

```
DROP_IT:
OB=ITEM(1)
O=Match(INV(0),OB) : Rem Find object number in list
If O<0 Then Print "But you don't have it " : Return
Rem Is Object worn?
If LOC(OB)=WORN
  Print "You'll have to remove it first" : Print : Return
End If
Rem Drop object
INV(O)=0 : Rem Set item to zero
Sort INV(0) : Dec HELD
Print "You dropped ";OBJECT$(OB) : LOC(OB)=R
Print
Return
INVENT:
Rem system command to display an inventory on the screen
APPARENT=0 : Print "You are carrying" : Print String$("-",16)
If HELD=0 Then Print "nothing" : Return
Rem Scan through the inventory list
For N=0 To 20
  O=INV(N) : Rem Get object number carried
  If LOC(O)=CARRIED Then Print OBJECT$(O) : APPARENT=1
Next N
If APPARENT=0 Then Print "nothing"
Print
Return
```

As they stand, these routines are only suitable for text-plus adventures. But the only difference between the text and graphics versions is the way the objects and messages are displayed on the screen. Everything else is identical.

It's simplest to call all the global and system commands from a single AMOS Basic routine. I've also executed the parser at the same time, just for good measure.

```
USER:
Rem This bit complements our hi-priority event
Rem and handles LOOK or DESCRIBE
If VERBOSE or VISIT(R)<2 or LOOK
  Gosub LOOK
End If
Gosub XPARSER:Rem Call up the parser!
Rem return if there's an error
If ER=1 Then Return
Print
Rem system commands are easy!
If ACTION=INV Then U=1 : Gosub INVENT
If ACTION=QUIT Then Stop
If ACTION=ADVSAVE Then U=1 : Gosub ADVSAVE
If ACTION=ADVLOAD Then U=1 : Gosub ADVLOAD
Rem global commands
Rem If the NOUN is CARD, then this system checks whether
Rem the preposition is GREEN or BLACK
Rem If so it loads the ITEM array with the appropriate object number
Rem It's used in lines like GET GREEN CARD or DROP BLACK CARD
If ITEM(1)=CARD and(PREP=PGREEN or PREP=PBLACK)
  If PREP=PGREEN : ITEM(1)=GREEN : End If
  If PREP=PBLACK : ITEM(1)=BLACK : End If
End If
Rem Handle LOOK
Rem Normally, they will also be special routines to handle
Rem specific commands like LOOK AT SHOE
Rem So this bit will only be entered if the earlier tests failed
If(ACTION=LOOK or ACTION=DESC or ACTION=EXAMINE)
  Rem If the user just types look, then we'll ignore the command
  Rem And let our LOCAL routine get on with it.
  Rem The SHORT and LONG description is selected as part of
  Rem The HI priority even routine I showed you in section 4.9
  If COUNT=1
    U=1
  Else
    Rem Otherwise, we print out a standard message
    Print "You see nothing special"
    U=1
  End If
End If
Rem I've placed these commands at the end, because I may want to test
Rem for special objects which could react when they are grabbed, or
dropped
Rem recognise PICK but no PICK UP
If ACTION=PICK and ITEM(1)<>0 and ITEM(1)<>UP Then U=1 : Gosub PICK_UP
If ACTION=DROP and ITEM(1)<>0 Then U=1 : Gosub DROP_IT
Return
```

This routine can be accessed from your room routine using a line like:

```
Gosub USER
```

Obviously we would need to define the various arrays at the start of our program as well.

4.10.3 Local commands

Some commands only make sense when the adventurer is occupying a specific location. A safe, for instance, can only be opened if the player is actually standing in front of it. As I've

previously indicated, it's best to use the same subroutine to handle both local events and commands. Each location in your adventure will have its own individual routine. This will control all the activities that can occur in the current room. Here's an example.

```
CELL:
R=1 : Gosub HI:Rem call up hi-priority events
Rem handle text descriptions
If SHORT
  Pen 4
  Print : Print : Centre(Border$("Padded cell",1)) : Print : Print :
Print
  Pen 2
Else
  Pen 4
  Print "You are a large padded cell, lined with lots of red rubber"
  Print "It's totally featureless except for a door to the west"
  Print "And yes, it's RED"
  Pen 2
End If
Rem Call PARSER and handle GLOBAL and SYSTEM commands
Gosub USER
Rem This system handles LOOK or DESCRIBE
If ACTION=DESC and COUNT=1 Then LOOK=1 : Goto CELL
Rem handle directions
If D=EAST and F(OPEN)=1 Then Goto SOUTH_CORRIDOR
Rem local commands
Rem STAND UP or GET UP
If(ACTION=STAND or ACTION=AGET) and ITEM(1)=UP
  U=1
  Bell
  If F(HEALTHY)=0
    Print "You stagger up from the floor and fall down flat on your
face!"
    Print "Your heel twists agonizingly as your shoe gives way."
    Print
  Else
    Print "You are now standing upright"
  End If
End If
Rem I know it looks horrible, but work it through
Rem This is SLIDE CARD IN SLOT
If(ACTION=SLIDE and PREP=IN and ITEM(2)=SLOT)
  U=1
  Rem SLIDE CARD IN SLOT
  If ITEM(1)=CARD
    Rem which one?
    If LOC(BLACK)=CARRIED and LOC(GREEN)=CARRIED
      Line Input "Do you mean the GREEN one or the BLACK ONE>";C$
      Rem Here's a mini-parser of special use
      C$=Upper$(C$)
      P=Instr("GREEN BLACK",C$)
    If P=0 : Print "HUH?" : Else ITEM(1)=CARD+P/7+1 : End If
    End If
  End If
  Rem SLIDE GREEN IN SLOT
  If ITEM(1)=GREEN
    Print "The slot digests the card, thinks a bit, and then"
    Print "spits it straight back at you!"
  End If
```

```
  Rem SLIDE BLACK IN SLOT
  If ITEM(1)=BLACK
    Print "The door emits a low agonised humming for thirty seconds"
    Print "and reluctantly slides open"
    F(OPEN)=1
  End If
End If
Gosub LO
Goto CELL
```

Note CARD, BLACK, SLOT, IN, GREEN and so on are all variables we've defined earlier in our program.

4.10.4 System commands

System commands allow the user to interact directly with the main program. As a rule, these should be defined using a Gosub routine rather than a procedure, since they usually have far reaching effects on the running of your program. Common system commands are LOAD, SAVE and VERBOSE.

Now for some examples of specific system commands:

```
ADVSAVE: Rem Save game
If COUNT=1
  F$=Fsel$("*.ADV","","Save an adventure")
  If F$="" : Print "Save game aborted" : Return : End If
Else
  W$=WORD$(1)-" "
  F$=Dir$+W$+".ADV"
End If
Print "Saving ";F$
Open Out 1,F$ : Rem save position and status
Print #1,R : Print #1,SCORE
Rem save flags
Rem FLAGS is a constant you've defined earlier
For N=0 To FLAGS : Print #1,F(N) : Next N
Rem save object locations
For N=0 To OCOUNT : Print #1,LOC(N) : Next N
Print "Ok" : Close 1
Return : Rem that's all
ADVLOAD: Rem Load game
If COUNT=1
  F$=Fsel$("*.ADV","","Load an adventure")
  If F$="" : Print "Loading aborted" : Return : End If
Else
  W$=WORD$(1)-" "
  F$=Dir$+W$+".ADV"
End If
If Exist(F$)=0
  Print "I can't find this file. Try again" : WORD$(1)="" : Return
End If
Print "Loading ";F$
Rem Load room
Open In 1,F$ : Rem open file for input
Input #1,R : Input #1,SCORE
Rem load flags
For N=0 To FLAGS : Input #1,F(N) : Next N
Rem load inventory
HELD=0 : Rem number of object held by character
```

```
For I=0 To 20 : INV(I)=0 : Next I : Rem zero inventory
Rem Load object locations
For N=0 To OCOUNT : Input #1,LOC(N)
  Rem load inventory
  If LOC(N)=CARRIED : INV(HELD)=N : Inc HELD : End If
Next N : Sort INV(0)
Rem close file
Print "Ok" : Close 1
Rem Zero out the visit array, to bring up a full description of each room
For L=0 To 100 : VISIT(L)=0 : Next L
U=0 : Rem set unrecognised word flag
Rem now jump to room routine using room number
ADV_LOC:
On R Goto CELL,SOUTH_CORRIDOR,NORTH_CORRIDOR,ALCOVE,DIAS
Rem This list should include the names of all your room
Rem handling routines in ascending order of their room number
Rem Whenever you add or remove a room, remember to update
Rem this line!
Rem If your adventure is large, split the line like so..
Rem If R<10 Then On R Goto ROOM1,ROOM2...ROOM9
Rem If R<20 Then On R Goto ROOM10,ROOM11..ROOM19
Stop
```

4.10.5 Movement commands

The movement commands allow the player to move through the various locations in the adventure. As I demonstrated in Section 4.5, they are usually implemented using a set of local commands defined separately for each particular room. The direction number is held in the variable *D*.

```
  If D=EAST Then Goto ROOM1
  If D=WEST Then Goto ROOM2
  : :
```

etc

4.10.6 Skeleton routine

Finally, here's a skeleton routine which provides a simple framework which can be fleshed out to create own your room routines.

```
ROOM:
Rem Set the LABEL to the name of your room
Rem set R to your room number
R=0 : Gosub HI
Rem Handle SHORT or long descriptions
If SHORT
 Print "Short description"
 Rem Or Display a picture
Else
 Print "Long Description"
 Rem Or display a picture
End If
Rem Interpret the users instructions
Rem execute GLOBAL and SYSTEM commands
Gosub USER
Rem Handle LOOK
If ACTION=DESC and COUNT=1 Then LOOK=1 : Goto ROOM:Rem change ROOM label
```

```
Rem Interpret directions. Change the commands to Goto the actual ROOMS
Rem In your game
If D=NORTH Then Goto ROOM1
If D=SOUTH Then Goto ROOM2
If D=EAST Then Goto ROOM3
If D=WEST Then Goto ROOM4
Rem Call up low priority events
Gosub LO
Goto ROOM:Rem Change to new room label
That's IT!
```

A full example is on the Game Maker's Companion disk from Sigma Press.

4.11 Conclusions

If like me, you're a fan of adventures, it's well worth the effort involved in writing one of your own. You'll probably find that the process of creating one is actually even more fun than playing someone else's.

I've already provided you with all the elements you need to generate your own adventure in a matter of hours. Many of these routines compare favourably with those included in dedicated adventure creators costing over £30. So there's now no reason why you can't produce a perfectly commercial adventure directly within AMOS Basic. After all, it's not often you get a perfectly legitimate chance to play God!

5

Role-playing Games

5.1 History

Role-playing games first emerged with the appearance of Dungeons and Dragons in the early 70s. D&D took a small group of people on an exploration of a vast imaginary world, where magic and adventure was a way of life. The aim was to try to simulate the action and excitement of a fantasy novel from the viewpoint of the people actually experiencing it.

One player, known as the referee, played the part of the story teller, and the other participants assumed the roles of the various characters in the plot. These characters were able to roam through the game world at will, solving puzzles, rescuing princesses, and killing the occasional monster. Whenever something interesting happened, such as a dragon appearing, dice would be rolled, and tables consulted to determine the final outcome.

Surprisingly enough, the market for these games turned out to be immense. So it wasn't long before a whole range of competitors sprang into existence. Some of these were based around a similar world of swords and sorcery as the original D&D. But other games branched out into different areas of fiction, such as SF and Crime. Typical examples of included Traveller, Super-heroes and Star Trek.

Over the course of time, D&D has been steadily evolving and improving. The latest *Advanced* version is now known as AD&D, and even this is currently undergoing a major revamp to prepare it for the 90s. Similarly, Traveller has been reborn in a new and even more exciting guise. Even the name's been changed! GDW have added Mega to create the amazing MegaTraveller system!

The main attraction of these role-playing games, (or RPGs for short) is that they are largely open-ended. Games can last from a couple of hours, to months or even years. Also, they are essentially a group activity, and there's a lot of fun to be had trying to out-fox your fellow players.

Unfortunately, it's often impossible to get a complete group of players around a table whenever we personally fancy a game. This has inevitably led to the production of special versions which can be played by a just single player.

Computers are ideal for this purpose, because they allow us to play out our favourite RPG from the comfort of our own living room. As yet, none of these games quite captures the flavour of the original role playing systems. But some of the better games, such as Dungeon Master, Eye of the Beholder and MegaTraveller, are starting to come impressively close!

In recent years, RPGs have managed to secure a respectable slice of the total software market. This fact has now been belatedly acknowledged by the original producers of D&D (TSR), who have released a whole series of *official* AD&D computer games for the Amiga. And GDW have entered the fray with the Zhodani Conspiracy for MegaTraveller fans. At the moment, the future looks pretty rosy for computer RPGs. Maybe they have finally come of age!

Incidentally, you may be confused about the difference between a role playing game and an adventure. An RPG can really be considered as a cross between an adventure and a war game. This lends a strong tactical element to it which is generally lacking in an adventure. There's also quite a bit of simulation work involved as well, in generating the results of the various actions and events.

If you've never played an RPG before, it's well worth searching around for a game called Hack which is now available from your favourite software library for around £3. Although it is incredibly crude by modern standards, and the graphics are practically non-existent, it remains an extremely enjoyable game, and can be highly recommended. I would also strongly suggest a look at FTL's incredible Dungeon Master series. These games only work on expanded machines, but have to be seen to be believed!

AMOS Basic is ideally suited for the creation of these role playing games. The combination of good screen animation, along with the powerful map definer accessory, makes RPGs well within the reach of the vast majority of Basic programmers. So providing you are prepared to invest enough time and energy into the project, there's no logical reason why you couldn't produce a game which compared favourably with anything on the market today.

5.2 Anatomy of a Role Playing Game

Before you can write a role playing game for yourself, you will need to understand a little about how they work. The basic principles are exactly the same for the computer and human versions.

The game revolves around the activities a number of characters, embarking on an exciting quest through a vast imaginary world. They can either be controlled by the player (player characters), or by the referee (Non Player Characters or NPCs). In order to avoid possible confusion, I'll start off by defining a couple of important terms.

From now on, **RPG** will refer to a computer game. **Real RPG** denotes the original Role playing systems which use human referees.

5.2.1 Character Classes

Most games allow us to generate characters belonging to a number of different professions, known as classes. Each class has its own specific range of abilities and skills. It's completely up to the players to decide which mix of characters are to be included in the current adventuring party.

In a fantasy game like D&D, the typical classes include:

Warriors	Fight the good fight
Magic Users	Provide a veritable stockpile of magical mayhem
Priests	Heal the parties wounds
Thieves	Find valuable hidden treasure
Knights	Do good deeds, rescue fair maidens
Assassins	Bring death for a price
Archers	Supply long range weapons

Similarly, space games such MegaTraveller split the characters into categories like:

Mercenaries	Just hi-tech warriors
Bounty Hunters	Seek out and destroy the enemies of the state
Star Troopers	The United Space Marines
Medics	Fix the damage caused by the previous three classes
Traders	Buy and sell cargo
Explorers	Boldly go where no adventurer has gone before
Pilots	Fly the spaceships
Engineers	Fix the damage caused by the Pilots!

The names of the classes naturally vary from game to game. But no matter what we call them, their essential qualities remain pretty much the same.

5.2.2 Attributes

Since we can't hold a real person's personality inside a computer, we'll need to devise a method of expressing the various strengths and weaknesses of our characters using simple numbers.

Each character is represented by a unique list of numbers known as attributes. These attributes determine the exact level of the characters' abilities, and can be tested to determine the success of failure of any given action in the game. As with the classes, the precise nature of these statistics will vary depending on our system. Common attributes are:

Intelligence/Brains/IQ

This measures how clever the character is.

Stamina/Endurance

Specifies the character's physical stamina. It affects how much damage the character can survive during combat, and how much weight he or she can carry.

Health/Hits-To-Kill

Holds a measure of the current health of the character.

Speed/Dexterity

How fast can the characters move? How quickly can they swing a blade?

Strength

How strong is the character? How hard are the character's blows?

Magic/Wisdom

Sets the overall strength and type of any spells a character is capable of casting.

Power/Psionic strength

Measures a character's current magical ability. Generally it will be temporarily reduced whenever a character casts a spell.

Rank/Social standing

Sets the position and authority of the character in the society we are creating.

Sex

This determines whether the current character is male or female. It's well worth sparing a thought to any lady adventures out there. Do we really want to continually refer to their female characters using words like *he?*

Normally, the higher the score, the better the attribute. The actual numbers are usually produced using a random number generator like the AMOS RND function. Each character class is generated independently. This avoids the possibility of having a magic user who is unable to cast any spells, or a fighter too weak to pick up his sword. I'll be discussing this problem in a little more detail later, when I'll be providing you with a simple character generator written entirely in AMOS Basic.

Additionally, most other objects such as weapons and armour will also possess attributes. The nature of these attributes will obviously be specific to the object in question. Assuming the values taken by an attribute could range from 1-20, we could define the character Olric thus:

NAME Olric
CLASS Warrior
STRENGTH 10
STAMINA 14
SPEED 15
INTELLIGENCE 5
MAGIC 3
POWER 2
RANK 0
SEX Male

These numbers represent the sum total of the abilities of the warrior Olric. They are collectively referred to as statistics (or stats for short).

Different attributes affect different things. Those for strength, stamina and speed can be combined to determine a character's ability for physical combat. This indicates the likelihood of a blow hitting Olric's opponent, and affects the amount of damage which would be inflicted if he hit the mark. Similarly, the attributes for power and magical ability determine Olric's ability to cast spells. As you can see, Olric is hardly suited for work as a magician.

According to his rank, Olric is currently the lowest of the low. So he's probably a beggar, or maybe even a politician! (Any advances on Manual Author?) But like most things in the RPG, the rank could easily improve during Olric's adventures. Another type of attribute measures the performance of a specific skill, such as safe-breaking or engineering. This can be represented either as a percentage chance to succeed, or as a numerical value which can be checked against a table. Here are some typical skills for you to examine:

Lock picking

This is a really useful talent in a dungeon type game.

Rifle shooting

If our game is based around crime, it might be a good idea to keep a separate measure of the character's shooting ability.

Spacecraft repair

Essential for some space games!

Invisibility

This is an ability which might well appear in a super-hero type game.

5.2.3 Experience

It's a well known fact that practice often makes perfect. The same is true for the characters in our RPG. Generally, all characters start off relatively puny, but as the game progresses their natural abilities slowly develop. This improvement is sometimes expressed in terms of the character's level. Each character is initially bestowed with a level number of 1, and an experience of zero. The experience is held in a simple variable, and can be increased whenever a character successfully surmounts an important obstacle in the game.

Once the experience exceeds a predetermined threshold, the level number is incremented by 1, and the appropriate attributes improved accordingly. Let's take another look at our trusty warrior Olric. At the present time, his main statistics are:

STRENGTH 10
STAMINA 14
SPEED 15
INTELLIGENCE 5
MAGIC 3
POWER 2
RANK 0

If Olric were to gain experience in fighting, and progress to level 1, each of the attributes of strength, endurance, and speed would be beefed up. This could be achieved by multiplying each of the three attributes by a appropriate level factor.

Using a level multiplier of 1.25, Olric's statistics would now be increased to:

STRENGTH 12
STAMINA 17
SPEED 18
INTELLIGENCE 5
MAGIC 3 (Not affected)
POWER 2 (Unchanged)
RANK 1

See how I've rounded the numbers down after the calculation. This process will be performed automatically by AMOS Basic. It's a direct consequence of using standard AMOS integer variables for our stats.

Note that if you do decide to use this system, you will need to be very cautious about the multipliers you choose for each level. These usually have to be worked out by careful experimentation, as there's no easy way of discovering them in advance. We'll need to play the game and check out the result.

Some games simplify things still further by assigning each level number with a name. For example:

NOVICE (Level 1)
APPRENTICE (Level 2)
MASTER (Level 3)

and so on. We can also keep a separate count of the success rate of every individual ability. So in addition to other attributes, a character could possess a range of optional skills like:

Lock-picking 50%
Star-ship navigation 30%
Horse riding 5%

As the character learns from experience, the percentage chances of successfully using a skill can be gradually improved. This approach is used to great effect by role-playing games like Runequest or MegaTraveller. Be warned though. It does require a whole load of extra variables for each attribute score. This inevitably adds to the complexity of the resulting program.

5.3 Scenarios

5.3.1 What's a scenario?

All Real RPGs are split into two separate components. The source book defines a detailed set of rules which form the mainstay of the game. These rules specify the precise mechanics of the gameplay, and are used to establish the outcome of the players' activities. There's also a great deal of essential background material. This establishes the history of the imaginary world which the characters are inhabiting. The scenario is equivalent to the plot, and provides full details about the particular story which will be enacted by the players. Due to the open-ended nature of these games, it is usually impossible to determine the final outcome of any given scenario.

One major difference between a Real RPG and the computer version, is that computer games are much more limited in scope. Unlike a human referee who can literally make things up as he goes along, the computer is forced to rely on a complete scenario which has been worked out in advance.

5.3.2 Setting the scene

Our first requirement is to devise a believable setting for our game. The game world can taken from a number of different genres, such as Fantasy (D&D and Bard's tale), Science Fiction (Traveller), Crime, and Super-heroes. Once we've decided on the background, we can then devise the individual scenario. The best source of inspiration is to revisit some of your favourite novels, or rent a few of the appropriate videos. However it's essential to avoid basing your game too heavily on a single work.

Nowadays, popular authors can expect to sell the computer rights to their works for very respectable sums. They are therefore unlikely to welcome you stealing any of the characters and situations wholesale out one of their books, without prior written permission. Comics in particular, are very sensitive about the names of their heroes being taken in vain. So any mention of a hero such as Superman in one of your programs might be very unwise!

There's also the question of originality. It's futile to rehash someone else's work, hoping no-one will notice. Even if you choose something really obscure, there will always be at least one person who is familiar with it. This is particularly likely in the field of science fiction, as many authors are now enthusiastic computer users.

But ignoring moral considerations, unless you invest the scenario with your own ideas, the game will be both derivative, and utterly boring. So try to think of something really different, which makes the game stand out from the crowd.

The purpose of the scenario is subtly to generate the feeling that the player is really experiencing the game world we have created. It's therefore essential to work out the precise rationale behind the various characters extremely carefully. Who are they? Why are they participating in the adventure? What are their individual goals?

5.3.3 Some basic scenario ideas

Thankfully, all scenarios can be roughly divided into a relatively limited number of basic plots. Here is a list of the more obvious possibilities:

- The purpose is to retrieve one or more valuable artifacts which are hidden somewhere in the game. (Used in Dungeon Master, Lord of the Rings and so on). There's often a pressing reason for this search, such saving the world or something, but this is normally totally incidental to the blood and gore!

- A dastardly villain is threatening the safety of the player's world. The characters are given the thankless task of destroying this horrible creature before it wreaks havoc. Personally, I'd run away and hide, but adventurers are suckers for punishment!
- A beautiful princess has been kidnapped by an evil monster. (or vice-versa!) The party of adventurers sets out bravely to rescue her, against incredible and (probably suicidal) odds. (This is a one-line summary of Star Wars.)
- The characters are a bunch of mindless psychopathic lunatics who want to kill as many monsters, and amass as much treasure as possible. (Who said we had to be original?)
- Our intrepid heroes are lost in the centre of a labyrinth. Their sole aim is to escape from this horrible predicament, before some unlikely monsters fancy a quick snack.
- The purpose of the game is to solve a complex logical puzzle of some sort (used in Ultima IV)

None of these plots is restricted to a single area. So the same idea can be used equally well in a science fiction game as in a sword and sorcery scenario. Furthermore, it's perfectly acceptable to use any combination of these ideas in our finished game.

5.3.4 The game map

The next step is to produce a complete map of the game world. It's best to start off with a rough sketch of this world on a piece of graph paper. Suitable materials can be purchased at little expense from any decent stationary shop. The advantage of working on paper, is that it places us under no constraints whatsoever about the eventual nature of the graphics. It also allows us to see the whole world at once, which would be very difficult if we attempted to draw the map directly with the TAME or TOME Map definers.

We can now carefully set up the various obstacles which will be arranged against the players, and choose the ensuing rewards. Sometimes, the only reward for the successful solution to a puzzle will be the satisfaction of beating the computer. On other occasions, the completion of a task will result in the discovery of an item of treasure, or in an increase in the abilities of the player's characters.

It is vital that we try to balance the risks taken by the characters with the possible gains. If our game fails to achieve this balance, it will either be monotonously easy, or impossibly difficult. In order to work, a role-playing game must also include potential solutions to any traps which are to be encountered by the player.

The best human referees adopt a hard but fair approach. And any game which kills off the characters randomly without due cause is unlikely to capture the player's imagination for any length of time. As a rough guide, I've listed some of the more common hazards our players might encounter, with a few of the possible rewards. These can be used as a valuable source of ideas when you are creating your game.

Hazards

Monsters
Events Pits; Falling walls; Booby-traps; Cursed weapons/treasure; Radiation.
Logic puzzles Hidden doors; Locked doors – Requiring specific key; Requiring brute force; Requiring a certain spell.
Teleport rooms As the players enter a room, they are instantly transported to a completely different part of the map. Don't inform the players about this immediately. Let them work it out for themselves. It's much more fun!
Switches/Levers Which need to be pulled in a particular order
Passwords Which need to be typed in
Objects Which need to be used in a certain way)

Rewards
Money
Weapons
Armour
Spells
Experience
Transportation Anything from a horse to a spaceship!

Special objects
Treasure
Keys
Maguffins These are objects needed to solve a particular puzzle. 3-2-1 fans rejoice!

We should now take our game map, and mark out the positions of the appropriate objects and puzzles. The location of the monsters can be either random, or thought up carefully during our initial design. If we're producing a set of rooms which will be encountered in a particular order, we should painstakingly tailor the type and strength of our problems to the expected abilities of the characters. After we have created our map, and generated the various hazards and treasure, our scenario will be complete. It's now time to start work on producing a computer game based on this scenario.

5.4 Game Plan of an RPG

I'll now discuss the detailed mechanics of a role-playing game. As always, I'll kick off with a bit of pseudo-code:

```
Initialise dungeon
If new game
then
  Generate characters
else
  Load saved game
endif
Repeat
    Input action
    If action is "save game" then save current dungeon
    If action is movement
    then
        check if movement is allowed
        If movement is legal
        then
            Move characters
            Redraw screen
        endif
    endif

    Check for an encounter
    If an enemy is encountered then Resolve combat

    Check for a trap
    If a trap is discovered then Spring trap

    If action is "Pick" then Get object
    If action is "Use" then Use object

Until Party is killed or Quest is completed
Terminate game
```

If you examine this description carefully, you should be able to isolate a number of the more important activities. These include:

Initialise game
Generate characters
Load game
Input Action
Save game
Check for legal move
Redraw screen
Check for an encounter
Check for a trap
Resolve combat
Spring trap
Get object
Use object

I'll be expanding some of these requirements in a little more detail during the rest of this chapter.

5.5 Creating a Character

5.5.1 Selecting the attributes

The choice of the possible character attributes can have far reaching consequences for the appeal of our game. Most RPGs have the following attributes in common.

Stamina (or Endurance) Resistance to attack
Speed (or dexterity) Speed of attack
Strength Force of a physical blow

Any other attributes can be invented completely at our own discretion. Here are few simple guidelines.

Fantasy games, such as D&D, need some measure of the character's magical ability. This results in attributes such as Wisdom, or Psionic strength. Similarly, Space games often require an indicator of a character's intelligence.

It's also useful to include an indication of whether the character is male or female. This allows female players to enjoy the game as much as anyone else, as it tailors the system messages to the appropriate gender. The work involved in this exercise is minimal, and anything which widens the appeal of our RPG has to be worthwhile.

The nature of our scenario may make certain skills essential for the completion of the game. If the adventure is futuristic, we may decide to include skills like engineering or astrogation. Unlike the attributes, these skills are entirely optional. Different characters can therefore have completely different ranges of skills. We can also choose a number of more unusual attributes. These can be used to *flesh out* the character's personality, and can add a whole new dimension.

Supposing our game involves several characters controlled by a single player. It is normally assumed implicitly that these characters will obey the player's orders without question. But what will happen if one of the characters is a coward? This character is liable to ignore any commands which place him in unnecessary risk.

So if the player tells him to attack a fire-breathing dragon with his bare hands, he is likely to run away in the opposite direction, terribly fast. That's what I'd do, anyway! We can

implement this technique by adding an extra attribute like Bravery. The Bravery of a character can either be displayed on the screen along with the character's statistics, or sneakily concealed for the internal use of our program.

This system is particularly effective if the party can grow during the course of a game. Some RPGs such as Ultima IV commence with a single character who can recruit help as the game progresses. In this situation, we can include a simple attribute denoting the trustworthiness of any character the player attempts to enlist.

Whenever the party now encounters some really valuable treasure, one of the characters might independently decide to steal it from the rest of the group. We could also envisage the party splitting up into several warring factions. This could have potentially hilarious results, especially if the baddies had hold of all the party's most potent weapons!

Obviously, this type of nastiness needs to be used with just a little caution. If the honour of a character is crucial to the game's solution, we'll have to provide some way for the player to work this out in advance. This might involve a magical artifact such as a jewel which turns a different colour depending on a character's integrity.

5.5.2 Generating a character

Whenever a game is played, a complete set of characters will need to be allocated to the player. The easiest possibility is allow the player to choose the party from a list of adventurers prepared in advance. The appeal of this idea, is that it avoids having to bother with a separate character generator.

While there is nothing inherently wrong with this approach, it is recommended that you give each character a distinct personality. Each character should be discussed separately in the documentation, which should explain precisely who they are, and why they are willing to embark on the current adventure. Careful use of this technique can add a feeling of atmosphere which is often lacking in randomly generated characters.

Alternatively, we can generate our characters afresh every time the scenario is played. This allows each player to create their own unique set of characters for the game. It also conceivably enables someone to adapt their favourite characters from a completely different game.

In a Real RPG, all our numbers would be produced by simply rolling a few dice on the table. But as normal dice are only capable of producing numbers in the range 1 to 6, it's extremely difficult to generate the precise ranges of numbers required for our attribute scores. So many RPGs use special polyhedral dice with up to 10 or 20 sides apiece. These are great fun to play with, and add a real air of tension to the game - *Roll me a twenty, or I'm dead!*. Magic!

Computer RPGs are forced to produce all the numbers mathematically using a random number generator. AMOS provides us with a built-in random number generator called RND for this purpose.

=RND (range)

Returns a random number from 0 to *range*. So RND(6) produces an unpredictable number between 0 and 6.

If we want to simulate a six-sided dice, we'll need to use a command such as:

```
D=RND(5)+1
```

This generates a score from 1 to 6, just like a standard dice.

RND(100) returns a percentage from 0 to 100. This percentage value is ideal for resolving our combat sequences, and can also be used with the skill system I mentioned earlier.

```
If LOCKPICK>Rnd(100)
  Gosub PICK_LOCK
Else
    Gosub FAIL
Endif
```

Here *LOCKPICK* is assumed to be an AMOS Basic variable holding the percentage chance of the character picking a lock.

Before using the RND command, it's a good idea to prime it with:

```
Randomize Timer
```

RANDOMIZE ensures that all the numbers generated are as random as possible.

Anyway, let's get back to the problem in hand. In order to generate our characters, we'll need to produce a list of statistics. If each attribute could range from 1 to 100, we could generate the main statistics of a character using a line like:

```
STRENGTH=Rnd(100):STAMINA=Rnd(100):SPEED=Rnd(100)
```

Unfortunately, this would yield a very poor spread of numbers indeed. Any character would be equally likely to get a strength of 99 as one of 9. So it would also be perfectly possible for a character to be generated with ridiculously low values in all three characteristics. Similarly, the program could accidentally bring forth a god-like character who could effortlessly destroy even the toughest monster with just a single blow. It's better to add an appropriate offset of some sort to the raw random number.

```
STRENGTH=rnd(10)+5:STAMINA=rnd(10)+5:SPEED=rnd(10)+5
```

This limits the initial stats of the character to values between 5 and 15. See how I've initially kept the attributes small. There's now plenty of scope for them to be increased as the character gains experience.

We can also combine several calls to the random number generator at once:

```
STRENGTH=rnd(5)+rnd(5)+rnd(5)+rnd(5)+4
```

This line is the exact equivalent to rolling four dice in a Real RPG. Most numbers produced by this function are clustered around an average of 14. The chances of getting an unusually high number such as 24 or an excessively low value are relatively small.

The big danger of this approach however, is that the random number generator used by AMOS Basic is not completely random. So if we're not careful, we can actually generate some very strange results with this system.

Here is a listing of a simple character generator for you to type in. Don't bother to enter the REM statements, as they are only included for documentation purposes.

Example 5.1: Character generation in AMOS Basic

```
Rem Assumes the character has the following 5 attributes
Rem Strength, Speed, Stamina, IQ, Magic. Each attribute can range from
1-100
C=1 : Rem current character
Rem I've made these arrays extra large for expansion purposes
Rem But only character 1 is used by this system
Dim STR(10),SPD(10),STA(10),IQ(10),MAG(10),NAME$(10),SEX$(10)
Dim TYPE(10),SPECIES(10)
Rem This stuffs for the various buttons and sliders
```

```
Dim SC(10,4),SVAL(10),SLIM(10,2),STEXT$(10,2)
Rem reads in the text for the sliders
For I=0 To 2 : Read STEXT$(I,1),STEXT$(I,2) : Next I
Data "Fighter","Human","Wizard","Dwarf","Priest","Troll"
Cls 0 : Curs Off : Paper 0
Reserve Zone 10 : Rem reserve some space for the zones
Rem draw up control panel
Gosub CPANEL1 : Gosub CPANEL2 : Gosub DISPLAY
Rem read control panel
Repeat
   ZN=Mouse Zone
   If Mouse Key
      Rem slider
      If ZN>4
         Gosub SEL_SET3 : Wait 10
      Else
         Rem button
         On ZN Gosub CSAVE,CKILL,CQUIT,CROLL
      End If
   End If
Until CBYE
Stop
Rem Create a character with current settings
CROLL:
Play 40,0 : Gosub CKILL : X=170 : Y=50 : L=10 : Gosub NINPUT :
NAME$(C)=S$
SEX$(C)="Male" : Rem choose sex
Text 168,74,"[M or F]?" : Repeat : X$=Inkey$ : Until X$<>""
Cls 7,170,68 To 250,76 : If(X$="F") or(X$="f") Then SEX$(C)="Female"
TYPE(C)=SVAL(1) : SPECIES(C)=SVAL(2)
Rem create basic species stats
On SPECIES(C)+1 Gosub HUMANS,DWARVES,TROLLS
Rem modify DISPLAY depending on profession
On TYPE(C)+1 Gosub FIGHTER,WIZARD,PRIEST
Gosub DISPLAY : Gosub STATS : Return
CSAVE:
F$=Fsel$("*.Asc","","Save a character") : If F$="" Then Return
Open Out 1,F$
Rem I know it would be simpler to print out a simple list of stats
Rem with PRINT#1,NAME$(C):Print #1,STA(C)....
Rem But I've hacked the routine so its compatible with the demo game
Rem on the example disk. To replace Olric, just add
Rem C=1 after NPLAYERS=1 in the routine MAKE_CHARACTER
Rem and then use MERGE ASCII to tag the character file to the end of
Rem the MAKE_CHARACTER module
Rem The Chr$(34) bit is the quote character "
Print #1,"NF$(C)="+Chr$(34)+NAME$(C)-" "+Chr$(34)
Rem Short name is the same as long name
Print #1,"NS$(C)="+Chr$(34)+NAME$(C)-" "+Chr$(34)
Print #1,"NTYPE$(C)="+Chr$(34)+STEXT$(TYPE(C),1)+Chr$(34)
Print #1,"NSPECIES$(C)="+Chr$(34)+STEXT$(SPECIES(C),2)+Chr$(34)
If SEX$(C)="Male"
P$="SEX$(C,0)="+Chr$(34)+"He"+Chr$(34)+":SEX$(C,1)="+Chr$(34)+"him"+Chr$(34)
   Print #1,P$
Else
P$="SEX$(C,0)="+Chr$(34)+"She"+Chr$(34)+":SEX$(C,1)="+Chr$(34)+"her"+Chr$(34)
   Print #1,P$
End If
Print #1,"Level(C)=0" : Print #1,"STR(C)="+Str$(STR(C))
Print #1,"STA(C)="+Str$(STA(C)) : Print #1,"HEALTH(C)=STA(C)"
```

```
Print #1,"SPD(C)="+Str$(SPD(C)) : Print #1,"IQ(C)="+Str$(IQ(C))
Print #1,"MAG(C)="+Str$(MAG(C))
Close 1
Return
CKILL:
Rem Kills off the current character
NAME$(C)=Space$(10) : TYPE(C)=0 : SPECIES(C)=0 : STA(C)=0 : SPD(C)=0 :
STR(C)=0
MAG(C)=0 : IQ(C)=0 : Cls 7,170,42 To 250,114+Text Base : Return
CQUIT:
Rem Quit
CBYE=True : Bell : Return
Rem examine characters
STATS:
Rem Print out statistics
Ink 2,7 : Text 170,50,NAME$(C) : Text 170,58,STEXT$(TYPE(C),1)+Space$(3)
Text 170,66,STEXT$(SPECIES(C),2)+Space$(4) : Text 170,74,SEX$(C)
Text 170,82,Str$(STR(C))-" "+Space$(4) : Text 170,90,Str$(SPD(C))-" "+Space$(4)
Text 170,98,Str$(STA(C))-" "+Space$(4) : Text 170,106,Str$(IQ(C))-" "+Space$(4)
Text 170,114,Str$(MAG(C))-" "+Space$(4) : Ink 2,6,1
Return
DISPLAY:
Rem Prints out title
Ink 2,7
Text 120,30,"Statistics" : Text 90,50,"Name     :":  Text 90,58,"Class    :"
Text 90,66,"Species  :" : Text 90,74,"Sex      :" : Text 90,82,"Strength :"
Text 90,90,"Speed    :" : Text 90,98,"Stamina  :" : Text 90,106,"IQ       :"
Text 90,114,"Magic    :" : Ink 2,6,1 : Return
CPANEL1:
Rem Creates buttons
Reset Zone
Cls 0 : Ink 7,0,2 : Set Paint 1 : Bar 80,20 To 250,190 : Ink 2,6,1
MAKE_BUTTON["Save",1,90,170] : MAKE_BUTTON["Kill",2,130,170]
MAKE_BUTTON["Quit",3,170,170] : MAKE_BUTTON["Roll",4,210,170]
Return
CPANEL2:
Rem creates sliders
LARROW$="<" : RARROW$=">" : TL=9
ZN=5 : AX=90 : AY=130 : BN=1 : Gosub MAKE_HSELECT : SLIM(BN,1)=2
TP=1 : Gosub SEL_DRAW3
AY=150 : BN=2 : Gosub MAKE_HSELECT : SLIM(BN,1)=2 : TP=2 : Gosub SEL_DRAW3
SEL_ST=4 : Rem start zone of slider bars
Return
Rem Generate dwarves
DWARVES:
STR(C)=Rnd(2)+4 : SPD(C)=Rnd(10)+8 : STA(C)=Rnd(15)+25
IQ(C)=Rnd(10)+5 : MAG(C)=Rnd(8)+8 : Return
Rem Generate Humans
HUMANS:
STR(C)=Rnd(5)+5 : SPD(C)=Rnd(10)+5 : STA(C)=Rnd(10)+20
IQ(C)=Rnd(10)+5 : MAG(C)=Rnd(5)+5 : Return
Rem generate trolls
TROLLS:
STR(C)=Rnd(4)+8 : Rem trolls are strong
SPD(C)=Rnd(5)+5 : Rem trolls are slow
STA(C)=Rnd(15)+30 : Rem trolls are also VERY tough!
IQ(C)=Rnd(3)+2 : Rem trolls are thick
MAG(C)=Rnd(2) : Rem lousy magicians
Return
Rem Generate Fighter
```

```
FIGHTER: STR(C)=STR(C)*1.5 : SPD(C)=SPD(C)*1.5 : STA(C)=STA(C)*1.5
Rem Fighters make lousy wizards
MAG(C)=MAG(C)/3
Rem Fighters are generally thick
IQ(C)=IQ(C)/2
Return
Rem Generate Wizard
WIZARD:
IQ(C)=IQ(C)*1.5+1 : MAG(C)=MAG(C)*2+1
STR(C)=STR(C)/1.5 : SPD(C)=SPD(C)/1.5 : STA(C)=STA(C)/1.2
Return
Rem priests
PRIEST:
IQ(C)=IQ(C)*2 : Rem priests are very well educated
MAG(C)=MAG(C)*1.5+1 : Rem priests are also fair magicians
STR(C)=STR(C)/1.2 : Rem not too strong
Return
Procedure MAKE_BUTTON[T$,ZN,X,Y]
   W=Text Length(T$) : H=8 : Rem Assumes 8x8 text font
   Bar X,Y To X+W+2,Y+H+2 : Rem Draw a box around the button
   B=Text Base : Text X+2,Y+B+2,T$ : Rem Position the button inside box
   Set Zone ZN,X,Y To X+W+2,Y+H+2 : Rem create a screen zone
End Proc
NINPUT:
Rem Alternative input command
Rem INPUTS: X,Y=Starting coordinates ,L=Maximum number of characters
Rem OUTPUTS: S$ = Holds the text entered by the user, padded to L chars
Rem Define string and set up fake cursor
S$=Space$(L) : CP=0 : CH=Text Length("M") : Ink 1,2 : Text X,Y," "
Repeat
   Rem read a character
   X$=Inkey$ : CS=Scancode : A=Asc(X$)
   Rem If the characters ok, use it
   If A>=32 and CP<L
      Rem Print character and load it into S$
      Ink 2,1 : Text X+CP*CH,Y,X$ : Mid$(S$,CP+1)=X$
      Rem Handle cursor
      If CP<L
      If CP<L-1 : Inc CP : R$=Mid$(S$,CP+1,1) : Else R$=X$ : End If
         If CP>0 : Ink 1,2 : Text X+CP*CH,Y,R$ : End If
      End If
   End If
   Rem Handle left cursor
   If(X$=Cleft$ or CS=65 or CS=70) and CP>0
      R$=Mid$(S$,CP+1,1) : Ink 2,1 : Text X+CP*CH,Y,R$ : Dec CP
      R$=Mid$(S$,CP+1,1) : Ink 1,2 : Text X+CP*CH,Y,R$
   End If
   Rem handle right cursor
   If X$=Cright$ and CP<L-1
      R$=Mid$(S$,CP+1,1) : Ink 2,1 : Text X+CP*CH,Y,R$ : Inc CP
      R$=Mid$(S$,CP+1,1) : Ink 1,2 : Text X+CP*CH,Y,R$
   End If
Until A=13
Ink 2,7
Return
MAKE_HSELECT:
Rem inputs bn=Bar number, ax,ay= coords of bar
Rem larrow$,rarrow$ = character strings used for Left and Right arrows
Rem get size of each button
X=AX+1 : W=Text Length(LARROW$) : Y=AY+1 : H=8
```

```
BW=Text Length(String$("M",TL))
Text X+2,Y+1+Text Base,LARROW$ : Rem draw in left arrow
Inc ZN : Set Zone ZN,X,Y To X+W,Y+H : Rem create zone around arrow
X=AX+W+BW+4 : Y=AY+1 : W=Text Length(RARROW$) : Rem get pos of right arrow
Text X+2,Y+1+Text Base,RARROW$ : Rem draw right arrow
Inc ZN : Set Zone ZN,X,Y To X+W,Y+H : Rem create zone around arrow
Rem draw selector
Box AX,AY To AX+(W+2)*2+BW+4,AY+H+2
Draw AX+W+4,AY+1 To AX+W+4,AY+H+2 : Draw AX+W+BW+4,AY+1 To AX+W+BW+4,AY+H+2
Rem save the coordinates of the box in the middle
SC(BN,1)=AX+W+5 : SC(BN,2)=AY+2 : SC(BN,3)=AX+W+BW+4 : SC(BN,4)=AY+H+1
Return
SEL_SET3:
Rem Read selection bar
BN=(ZN-SEL_ST)/2 : D=ZN mod 2
If D=1
   If SVAL(BN)<SLIM(BN,1) : Add SVAL(BN),1 : Gosub SEL_DRAW3 : End If
Else
   If SVAL(BN)>0 : Add SVAL(BN),-1 : Gosub SEL_DRAW3 : End If
End If
Return
SEL_DRAW3:
Rem inputs: bn=bar number,stext$() holds bar text
Rem sc() holds bar coordinates,bw=width of centre area
S=SVAL(BN) : L=Text Length(STEXT$(S,BN))
Cls 6,SC(BN,1),SC(BN,2) To SC(BN,3),SC(BN,4)
CW=(BW-L)/2 : Text SC(BN,1)+CW,SC(BN,2)+Text Base,STEXT$(S,BN)
Return
```

I've divided the possible characters into three species: Humans, Dwarves and Trolls. But we could quickly modify these races for use with a SF game. All we would need to do is omit the magic, and change the species names to something really alien. So the trolls might by called Zaneck.

Each race has its natural advantages and its inherent disadvantages. This preserves the balance of the game and avoids generating characters which are too powerful to play realistically.

It's best to start off with a good idea about the type of races we are trying to produce. Here are some potted descriptions of the races I defined in Example 5.1

Humans: The most flexible of the character types. They are powerful enough to reasonably good at magic, yet if they have the inclination they make formidable fighters.

Dwarves: Magic experts. They are very tough and fast, but they are not particularly strong. So a dwarf would have to work very hard to become a successful fighter.

Trolls: The fantasy equivalent of tanks, these people are natural born killers. They may not be too bright, but they a VERY hard to kill.

The generation system for these using can be found in the routines starting at labels HUMAN, DWARF and TROLL.

Once the players have chosen a race for their character, they need to select a profession. In Example 5.1, I've provided just three possible classes: Fighter, Wizard and Priest. Again, we could easily change these to reflect any scenario we had in mind. We would simply need to change some of the attributes, and fiddle around with the figures.

If we are allowing for the possibility of female characters, we might decide to modify these attributes to compensate for the strengths and weaknesses of each sex. We could, for instance decide to reduce the Strength values of female characters. Providing we compensated with a hefty boost of the character's Speed, all would be well. Personally though, I prefer to use the same systems for ALL characters no matter what their sex, as it's both simpler and fairer. It's entirely up to you.

As with races, every class is generated by a separate subroutine in our Basic program. Each routine modifies the raw attributes to reflect the result of years of training in a particular skill. I've implemented this alteration using a system of multipliers. So the more talents an individual has in a certain profession, the higher their eventual abilities will reach after training.

A wizard might be generated using the lines:

```
WIZARD:
IQ(C)=IQ(C)*1.5+1: MAG(C)=MAG(C)*2+1
STR(C)=STR(C)/1.5: SPD(C)=SPD(C)/1.5:STA(C)=STA(C)/1.2
```

This would normally produce a physically weak character with low Speed and high Magical ability. Perfect for a wizard. Similarly, a fighter would need high values for the Strength, Speed and Stamina. Look at the FIGHTER: routine for an example of this type of system.

Note that I derived the numbers used by this generation system purely by experimentation. There's no guarantee that the same values will be especially suitable for a specific game. So it's important to experiment with these generators for yourself, in order to produce a character creation system which is just right for your particular game.

5.5.3 Managing the characters' possessions

Each character in the RPG will come equipped with a number of valuable possessions, which will hopefully increase in number during the course of the game. These items can be represented by simple digits, and can be manipulated using the same inventory systems I showed you in Chapter 4. At the start of our program, we'll need to define several arrays to hold our various item lists.

```
Dim SACK(10,40)
```

The SACK array stores all the items carried by the current party. Our present definition is capable of holding the item numbers of up to 40 objects for each of 10 possible characters. Each item is assigned an individual identification number. This can be used to find the attributes of our objects from some sort of table.

```
Dim OBJECT$(N)
```

Holds the name of each item in the RPG.

```
Dim OTYPE(N,max_attributes)
```

OTYPE stores the attributes of *N* possible objects; *max_attributes* holds the maximum number of statistics used by each object. These attributes can be devised entirely at our own discretion, but might include any of the following:

Weight: How heavy is the object? Each character will have a definite limit to the amount of goods they can carry, usually determined by their Strength.

Size: Is the object as small as a pin, or as large as an elephant? It'll make a heck of difference when our characters try to lug it around.

Worth: How valuable is the object? Can the players sell the item at a profit, and use the money to buy better weapons and armour?

Magic: Is the object magical in some way? If so, the Magic number should be set to a number representing the type. This can be subsequently used to choose between several Magic routines using an ON..GOSUB:

```
On MAGIC gosub RING,WAND,ORB,AMULET
```

where RING, WAND, ORB and AMULET are assumed to be predefined routines to handle the various item types.

WEAPON

If the object is a weapon, we'll also need to include the weapon number as well. We can use the value to find the effectiveness of the item in the WEAPON array. I'll be explaining how this works in Section 5.10.4 (*Combat*).

Finally, we'll need an array to hold each item's position in the RPG. This can be achieved using a single OPOS array like so:

```
Dim OPOS(N)
```

Our item coordinates can now be combined together into a single large value,using the following formula:

item position=(Y coordinate * number of columns)+ X coordinate

We then check for an object using an inventory system like the one in Chapter 4, or include it directly in our game map. An example of this technique can be found in Section 5.8.4 (*Identifying a monster*)

5.6 Drawing the Map

Unlike adventure games, a map of the current surroundings of the party is essential in any true RPG. This is because all role-playing games incorporate a strong tactical element, which necessitates a display of the type and position of the various monsters.

Over the years, several approaches have been used for these displays. In this section, I'll briefly examine some of the more popular ones, and explain how they can be implemented using AMOS Basic.

5.6.1 Displaying a map from above

This is the undoubtedly the easiest method of viewing the terrain around the party. Maps can be created using the standard TAME map editor, or the AMOS TOME utility, available separately from Shadow Software. You can also use the crude, but serviceable, TINY_TAME routine supplied free with the Game Maker's manual. See Example 5.2 for a first tantalising glimpse of this remarkable little utility.

Since I'll be explaining these maps in detail in Chapter 7 (*Screen Scrolling Techniques*), I'll currently limit myself to a brief explanation of the general principles, and a few large examples to get you started!

Our map is divided into a number of small images called icons. Icons are special images held in memory bank 2 which can be displayed on the screen at high speed using a simple PASTE ICON command from AMOS Basic. They're rather similar to sprites or Bobs, except that there are no facilities for moving them across the display.

Like Bobs, every icon image is assigned its own identification number starting from 1. The images can be created in three ways.

- If we're using the advanced TOME system, we'll be able to draw each icon directly from the map editor. Easy eh?
- Other users can draw their icons as if they were sprite or Bobs, with the standard AMOS Sprite editor. They can then convert these images into the equivalent icons using a Icon converter on the AMOS program disk. Note that the latest version of the Sprite Editor (Spritex) also includes a built-in icon definer. It's provided free with the AMOS Compiler.
- If, like me, you need to create icons inside one of your programs, you can grab them straight from the screen using the GET ICON command from AMOS Basic.

In order to distinguish our map icons from the ones we are using for our control panels, we'll give them a special name. From now on, tiles are icons used to create our map areas, and icons are used to create the buttons in our control panels. Ok?

Each tile portrays a particular feature of the terrain, such as a wall, a hill or a tree. These features can be combined to generate a large game map on the screen.

The map is divided into an arrangement of rectangular slots, each of which contain the number of a single tile. Figure 5.1 contains a simple demonstration of this idea. It creates are large block of tile number 1 surrounded by a border of tile 2.

Remember that these tiles each hold a separate image from the icon bank. So tile 1, for instance, might represent the walls of a castle, whereas tile 2 could depict one of the paving stones. Depending on our tile images, the map we've defined in Figure 5.1 could look extremely neat.

This map data is stored in its own separate memory bank (usually 6). The size is entirely up to us, but it can certainly be far greater than a single screen. We can therefore fit an entire level of our RPG into a single map.

	0	1	2	3	4	5	6	7	8	9	10	.....								
0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
3	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2
4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Figure 5.1: A simple game map

What's more, because we are only storing the numbers of our icons, these maps use tiny amounts of memory. A 100x100 map for instance, would consume just 10K. Of course, we'd also need to take account of our various images used by our tile data. But since we are reusing these images again and again, we can create massive game maps in a fraction of the normal memory. Here's a demonstration of the system in action. It's very crude, but don't worry, I'll be expanding it later!

Example 5.2: Drawing a map

```
Rem Map view 1. It's crude, simple, but it demonstrates the idea
Rem Activate memory conservation system
Close Workbench : Close Editor
Rem Activate double buffering
Double Buffer
Rem Open a separate screen to enter our keyboard commands
Screen Open 1,320,30,2,Lowres
Rem Prepare for a long wait
Curs Off : Cls 0 : Centre "<BUILDING MEMORY BANKS>"
Rem Use screen 0
Screen 0
Hide On : Flash Off
Curs Off : Cls 0
Rem Only use values of 16 and 32 for twidth and theight
TWIDTH=16 : THEIGHT=16
Rem Make some tile images
MAKE_ICONS[TWIDTH,THEIGHT,0]
Rem load our map into memory
Gosub MAP_INIT
Rem set position of viewing area on map
Rem This MUST be in multiples of a single icon
Rem So all widths and heights should divide evenly by sixteen
Rem Otherwise, you can move it anywhere you like
Rem TVIEW initialises my map drawing routine
TX=80 : TY=32 : BX=240 : BY=160 : Gosub TVIEW
Rem Draw first section of the map with TDO
Rem MX and MY are the map coordinates
Rem They are measured in units of a single tile
MX=0 : MY=0 : Gosub TDO
Ink 6 : Box TX-1,TY-1 To BX+1,BY+1
Pen 2 : Paper 0
Rem Zap screen 1
Screen 1 : Cls 0 : Screen 0
Rem Turn off autoback system
Autoback 0
Rem Main loop
Do
  Gosub ENTER:rem call up a routine to move around the map
  Rem Gosub CURSOR (see later)
Loop
ENTER:
Rem Enter coordinate values on screen 1
Screen 1
Cls 0
Locate 0,0 : Input "Enter X coordinate of map:-";MX
Locate 0,1 : Input "Enter Y coordinate of map:-";MY
rem flick back to screen 0
Screen 0
Rem Check for a legal move
If MX>=0 and MY>=0 and MX<=MPX-SWIDTH and MY<=MPY-SHEIGHT
  rem if so, redraw screen at new position
  Gosub TDO
Else
  rem error message
  Screen 1
  Bell
  Locate 0,2 : Print "Coordinates outside map" : Wait 100
  Locate 0,2 : Cline
```

```
  Screen 0
End If
Return
Procedure MAKE_ICONS[W,H,S]
Rem This routine makes some icon images for our map
Rem W,H are the width and height of the images
Rem S is the number of the first tile to be created
  Cls 0 : Pen 15
  Get Icon 1,0,0 To W,H
  Rem make some blocks
  For B=2 To 5
    Ink B+4 : Bar 0,0 To W-1,H-1
    Ink 2 : Box 0,0 To W-1,H-1
    Get Icon B+S,0,0 To W-1,H-1
    Cls 0,0,0 To W+1,H+1
  Next B
  Rem make some diamonds
  For B=6 To 9
    Ink B
    Polygon W/2-1,0 To W-2,H/2-1 To W/2-1,H-2 To 0,H/2-1 To W/2-1,0
    Get Icon B+S,0,0 To W,H
    Cls 0,0,0 To W+1,H+1
  Next B
End Proc
Procedure TINY_TAME[T$,W,H]
Rem Creates a map from a string
Rem Don't worry about how it works. I'll be explaining it in chapter 7.
  Erase 6 : Reserve As Data 6,W*H+50
  Doke Start(6),W : Doke Start(6)+2,H
  Rem load map
  For M=0 To Len(T$)-1
    I1=Peek(Varptr(T$)+M)
    If I1=32 Then IC=0
    Rem handle numbers (All TILE values are ONE LESS than the icon nos)
    If  I1>=Asc("1") and I1<=Asc("9") Then IC=I1-Asc("1")
    Rem handle letters (A=10,B=11..etc)
    If I1>=Asc("A") and I1<=Asc("Z") Then IC=I1-Asc("A")+10
    Rem I'm assuming that A and a are the same!
    Rem Feel free to change it if required - Just add 36 to IC instead of 10
    If I1>=Asc("a") and I1<=Asc("a") Then IC=I1-Asc("a")+10
    Rem check for an error
    If IC>Length(2) or IC<0 Then IC=0
    Poke Start(6)+M+4,IC
  Next M
End Proc
TVIEW:
Rem This routine sets up the window for a TOME compatible map
S6=Start(6) : MPX=Deek(S6) : MPY=Deek(S6+2)
TWIDTH=Max(16,TWIDTH) : THEIGHT=Max(16,THEIGHT)
W=Abs(BX-TX) : H=Abs(BY-TY)
Rem Get window height in tiles
SWIDTH=W/TWIDTH : If W mod TWIDTH<>0 Then Inc SWIDTH
SHEIGHT=H/THEIGHT : If H mod THEIGHT<>0 Then Inc SHEIGHT
Return
TDO:
Rem This routine draws section of map in the window set by the TVIEW routine
Rem The drawing starts at map co-ordinates MX,MY
S6=Start(6)+4
If MX<0 or MY<0 Then Return
Rem If MX<0 or MX>MPX or MY<0 or MY>MPY Then Return
```

```
YS=MY*MPX
Rem SHEIGHT=Height of the viewing window in tiles
For Y=0 To SHEIGHT-1
  XS=MX
  Rem SWIDTH=WIDTH of the viewing window
  For X=0 To SWIDTH-1
    BL=Peek(S6+(XS mod MPX)+YS)
    Paste Icon TX+X*TWIDTH,TY+Y*THEIGHT,1+BL
    Inc XS
  Next X
  Add YS,MPX
Next Y
Screen Swap 0 : Wait Vbl : Screen Copy Physic(0) To Logic(0)
Return
TVAL:
Rem This returns the icon number at Map Coordinates X,Y
Rem X=X coord :Y=Y coord:T=Tile (Icon) number
S6=Start(6)+4
YS=Y*MPX : T=Peek(S6+YS+X)+1
Return
TPLOT:
Rem Inserts an icon into the map at coordinates X,Y
Rem X=X coord :Y=Y coord:T=Tile (Icon) number
S6=Start(6)+4
YS=Y*MPX : Poke S6+YS+X,T-1
Return
MAP_INIT:
Rem Loads the map using TINY_TAME; Set up the string to hold the map
T$=""
Rem It's essential to make sure that you get the length and height
Rem Exactly right. Otherwise the map will look impossibly weird
Rem This line is a ruler!
Rem Just run your fingers across the number keys five times
Rem    01234567890123456789012345678901234567890123456789
T$=T$+"22222222222222222222222222222222222222222222222222":Rem 50 cols
T$=T$+"2                                                  2":Rem 24 rows
T$=T$+"22222222222                33333333333333333  2"
T$=T$+"22222222222    66          33333333333333333  2"
T$=T$+"22226622222   6666                       33  2"
T$=T$+"23366663333    66           66666         33  2"
T$=T$+"23366663333                666666666     33  2"
T$=T$+"23336633333    88           66666         33  2"
T$=T$+"23333333333   8888                       33  2"
T$=T$+"2  3333        88    333333333333333333333333   2"
T$=T$+"2   33               333333333333333333333333   2"
T$=T$+"2   33               33       55555          8  2"
T$=T$+"2   33               33       555555555555555588 2"
T$=T$+"2   33      77       33          55     55   8  2"
T$=T$+"2   33     7777      33         666666  55      2"
T$=T$+"2   33      77       33         666666  55      2"
T$=T$+"2   3333333333333333333         666666  55      2"
T$=T$+"2   333333333333333333                  55      2"
T$=T$+"2           33          55              55      2"
T$=T$+"2           33     555555555555555555555       2"
T$=T$+"2           77     555555555555555555555       2"
T$=T$+"2         777777                               2"
T$=T$+"2        77777777                              2"
T$=T$+"22222222222222222222222222222222222222222222222222"
TINY_TAME[T$,50,24]
Return
```

Example 5.2 builds up a simple 50*24 map, and allows us to display it on the screen a chunk at a time. All coordinates are measured in units of a single tile. So 10,5 refers to a point ten tiles right and five tiles down.

Here's a quick run-down of the drawing system:

Variables:

TX,TY,BY,BY contain the screen coordinates of the viewing window. This can be more or less anything we like.

TWIDTH and *THEIGHT* hold the width and height of each tile in our map.

MX,MY choose the part of the map which will be drawn on the screen. 0,0 represents the top left and corner of the playing area.

Routines:

MAKE_ICONS makes a series of nine tile icons for use as our building blocks.

MAP_INIT loads our map from a definition string, and saves it into memory bank six using the TINY_TAME procedure.

TINY_TAME[T$,W,H] converts a map from a string of characters into a TOME compatible memory bank. Each character represents the state of a single tile icon in the final map. Characters 1 and " " select tile 1, which is normally used for the floor. Characters 2 to 9 refer to the equivalent tile numbers. If we've more than nine tiles, we can also use the letters A to Z: A selects icon 10, B=11 and so on up to Z (tile 36).

Try changing the map definition at MAP_INIT for yourself. Select the Insert button from the Editor window, and type away using the keys 2 to 9. When you've finished, click on Overwrite to get the keys back to normal. With practice, you should be able to create some quite interesting maps with these tiles.

TVIEW initialises the MAP drawing system and defines a few important variables. TDO redraws a new section of the map starting from *MX,MY*.

I've also defined the following couple of routines for your own use:

TPLOT plots tile number *T* at position *X,Y*. It has no effect on the current display. TVAL reads the icon number at map coordinates *X,Y*, and loads it into the variable *T*. I'm not exploiting TVAL or TPLOT in the current program. But they'll come in rather handy later.

5.6.2 3D effects

Another possibility is to draw our displays in glorious 3D. Although 3D effects can be quite difficult to create, they can certainly provide the perfect backdrops for our games. Due to the complexity of these routines however, I don't have the scope to cover the detailed 3D programming system here. I'm afraid there wouldn't be enough space for the rest of the book! But never fear, help is at hand. I've thoughtfully included a large, and vaguely interesting example on the Game Maker's Manual companion disk. This also contains full listings of all the example programs in this book. So it could save you hours of typing! I've even supplied a few other routines which didn't quite make the final cut. Contact Sigma for details.

Alternatively, if you're a fan of the TOME map definer, you'll be interested to hear of a special Dungeon disk from Shadow Software (and the AMOS User group). This costs just £5, and provides the routines to convert any TOME map into an impressive 3D dungeon. It also includes some impressive wall images created by Adam Fothergill. It's almost up to Dungeon Master standards!

Games such as FTL's Dungeon Master were certainly not written overnight. Frankly, you would have to work long and hard to achieve an effect which even vaguely approached this excellent game. Despite this, 3D graphics are certainly worth thinking about, especially for the combat sequences. It's quite possible to combine a standard 2D system for mapping the dungeon, with a powerful 3D combat sequence using animated Bobs.

5.7 Controlling the Character

After we've created our maps, and implemented them using AMOS Basic, we will be ready to add the various puzzles and combat sequences. The nature of these routines will vary from game to game. One module which is crucial to all games however, is the routine which reads and interprets the player's commands.

5.7.1 Entering our command from the keyboard

Regardless of the availability of the complex input devices like the mouse and the joystick, the Amiga's keyboard is still a popular way of entering the commands in an RPG. This is because, unlike other input devices, there is no real limit to the amount of information which can be entered quickly and easily by the player.

Movement can be implemented using either compass directions such as *North, South, East* and *West*, or the standard Amiga's cursor keys. Other commands, like *Fight* or *Unlock* can be entered from the keyboard as in a normal text adventure. Each command will have its own particular key combinations.

When we are designing these codes, we should make a conscious effort to keep them as short and simple as possible. This is especially true of the movement and combat options. Nowadays, few people will be happy to type in *North* or *South* every time they move.

If we are using single key commands, we'll need to choose keys which will be easy for the player to remember. So the *Get object* command should be controlled using g or p rather than z. We can also use the Amiga's function keys. These provide us with an ideal way of entering instructions. If we are intending to type a long command, we can employ the familiar INPUT command as normal. But if we want to read a single keypress, we'll need to make use of some of AMOS's more advanced functions. AMOS Basic provides us with a whole host of commands for this purpose.

k$=INKEY$

INKEY$ (pronounced INKIES) is a standard Basic command which reads a single character from the keyboard. If a character isn't available, then an empty string "" is returned. Note INKEY$ doesn't echo the character on the screen. So unless we display it with PRINT, the keypress will have no visible effect.

Here's an example:

```
Do
  K$=INKEY$
  Print K$;
Loop
```

Try typing some random characters from the keyboard, and see what happens. Also try the cursor arrows. As you can see, the text cursor moves automatically in the appropriate direction. The cursor keys actually return special codes which can be printed directly on the screen to position the cursor bar anywhere we wish. Interestingly enough, these codes are also available from the functions CLEFT$, CRIGHT$, CUP$ and CDOWN$. This provides a simple way of testing the cursor keys in our programs. Look at the following fragment of code:

```
K$=Inkey$
If K$=Cup$ Then Gosub UP
If K$=Cdown$ Then Gosub DOWN
If K$=Cleft$ Then Gosub LEFT
If K$=Cright$ Then Gosub RIGHT
```

Here UP, DOWN, LEFT and RIGHT are assumed to be the routines we are using to handle the various movement directions. Here's a practical demonstration of this idea, which can be substituted for the ENTER routine in Example 5.2.

```
CURSOR:
X$=Inkey$ : If X$="" Then Return
OX=MX : OY=MY
If X$=Cleft$ and MX>0 Then Dec MX
If X$=Cright$ and MX<MPX-SWIDTH Then Inc MX
If X$=Cup$ and MY>0 Then Dec MY
If X$=Cdown$ and MY<MPY-SHEIGHT Then Inc MY
If MX<>OX or MY<>OY
  Gosub TDO
  Screen 1
  Locate 0,1 : Print "X coordinate :-";MX;"  ";
  Locate 0,2 : Print "Y coordinate :-";MY;"  ";
  Screen 0
End If
Return
```

As promised, INKEY$ also allows us to enter our individual keyboard commands. In this case, we simply grab a character into a string variable, and test it for a valid order using the INSTR function I showed you in Chapter 4.

=INSTR(c$,s$)

INSTR searches for the first occurrence of the character *S$* in the command string *C$*. If the test is successful, then the position of *S$* is returned directly, otherwise we get a value of zero. So:

```
Print INSTR("AMOS","A")
```

returns 1, while:

```
Print Instr("AMOS","S")
```

prints a value of 4. Since all our commands are just single letters, we can pack them straight into a test string like so:

```
C$="AMOS"
```

Each letter in AMOS now represents a different command from the keyboard. We can now use INSTR to get a unique identification number for our order.

```
N=INSTR(C$,K$)
```

Supposing we'd pressed the letter O. This would be checked against the commands in *C$*, and found at position 3. Once we've recognised our command, we can use the order number to jump straight to the appropriate routine with the ON GOSUB instruction:

```
If N<>0 Then On N Gosub ATTACK,MOVE,ORDER,SEARCH
```

Here's a small example of this system in action. Just to keep thing simple, I've converted the order number into a NOTE. You've bought the software, read the book, now's your chance to play the tune!

```
Rem C$ is the command string.
Rem It contains a simple list of all your orders
C$="AMOS"
Do
    Rem UPPER$ converts the number into UPPER case
    Rem So a and A return exactly the same value (A)
    K$=Upper$(Inkey$)
    Rem Echo the letter on the screen
    Print K$;
    Rem Find the letter in the command string C$
    C=Instr(C$,K$)
    Rem If the command is recognised, then play a note
    Rem Otherwise do nothing
    If C<>0 Then Play C*5,C*5
Loop
```

If we are using this system to enter vital commands, we can occasionally encounter problems with the keyboard. Normally, whenever we press a key, the character repeats after a short time. If we're not careful, this could lead to a stream of commands when we only intended to enter just one. The easiest solution is to make use of the KEY SPEED command like so:

KEYSPEED lag, speed

lag sets the amount of time a key has to be pressed before the repetition starts. All timings are measured in 50ths of a second. *speed* changes the delay between each character as it's echoed onto the screen.

The default values are around 10,3, but we can reduce the repeat rate to a crawl with:

```
keyspeed 100,100
```

Use `keyspeed 10,3` to get back to normal.

Sometimes however, KEY SPEED isn't enough. During critical activities such as combat, we need to guarantee that each order requires as separate keypress.

AMOS automatically keeps a list of all our recent keypresses in a special memory area known as the keyboard buffer. We can erase this buffer at any time using the CLEAR KEY command. If we call this instruction just before testing the keyboard, we can force the player to enter a new character before responding. This technique is used to great effect in the combat simulator I'll be showing you later.

Another problem with INKEY$ is that it's only able to test the keys which return an actual character value. Some keys, such as F1, or HELP don't! We can however, read them using the SCAN CODE and KEY STATE functions from AMOS Basic.

=SCANCODE

Each key on the Amiga's keyboard has its own unique identification number ranging from 0 to 255. After we've tested the keyboard with INKEY$, we can get at the invisible scancode with a simple call to the SCAN CODE function.

Example:

```
Do
  K$=Inkey$ : S=Scancode
  If S<>0
    Locate 0,0
    Print S;" ";
  End If
Loop
```

As with INKEY$, try hitting a few keys at random. The equivalent scan code will now be printed on the screen, and you can jot down the numbers of the ones you want. In particular:

- The function keys start at 80 (F1), and continue through to 89 (F10)
- HELP has a scan code of 95
- ESC = 69
- TAB = 66

If we need to repeatedly test a single key such as HELP, we can use the KEY STATE function like so:

```
If Keystate(95) Then Print "H..E..L..P"
```

=KEYSTATE(scancode)

KEYSTATE checks the key specified by the scancode and returns a value of TRUE (-1) if it's currently being pressed, otherwise FALSE(0).

5.7.2 Controlling the characters with the joystick

The Amiga's joystick is very useful for entering movement commands and for controlling the character during combat. I'll concentrate on the problem of movement, as this is of vital importance to the design of our games. I'll be providing a complete demonstration of this system in Example 5.3. But we'll start off with a bit of general theory.

The movement process can be illustrated with the following fragment of pseudo-code:

```
Get movement direction
If move is legal
  If character moves off the screen
    Then
      Display the next section of the screen
      Reposition character at edge of screen
    Else
      Redraw character at new position
    Endif
  Endif
Endif
```

This system only redraws the surroundings when the character crosses the edges of the current screen. It's known in the trade as a push scroller. In order to write one, we'll need to create separate routines for each of the following activities:

Get movement direction
Check for a legal move
Display the next section of the screen
Reposition character at edge
Redraw character
Let's have a look at each of these activities in turn.

Get movement direction

This is easy, as we can read the required commands straight from the joystick using the JLEFT(), JRIGHT, JUP and JDOWN functions I showed you in Chapter 3. These return a value of either TRUE(-1) or FALSE(0) depending on whether the joystick has been tugged in the appropriate direction.

Check for a legal move

Hmm.. Before we can proceed, we'll need to decide precisely what we mean by a legal move. Here's a definition for you to try out.

"A legal move is one which lies within the current map and does not attempt to occupy an existing object such as a wall or another character"

It's simple enough to check whether the move is inside the map. The TVIEW routine in Example 5.2, defined two variables for just this purpose:

MPX holds the width of the map in tiles

MPY contains the height of the map.

Our character coordinates are usually split in two parts:

CX,CY indicate the position of our character relative to the top left corner of the map display. These coordinates will be used by our drawing routines to reposition the character after a successful movement.

MX,MY hold the starting point of the visible section of the map. They're needed to redraw the map after the player has crossed the edge of the screen window.

The final map coordinates of the character are therefore:

CX+MX,CY+MY

So we could perform our test with a line like:

```
If CX+MX>=0 and CX+MX<MPX and CY+MY>=0 and CY+MY<MPY
   Then
     Gosub OK
   Else
     Gosub ILLEGAL
   Endif
```

The equivalent routine in Example 5.3 looks slightly different, as I've optimised it for extra speed. But the principles are exactly the same.

Checking for an occupied map location is a bit more tricky. We'll need to get the tile number at coordinates CX,CY and then compare it against a list of the tiles denoting walls, floors, objects and characters. In example 5.2 I thoughtfully included a simple routine called TVAL, which returns the tile number directly. It's used in our program like so:

```
X=CX+MX:Y=CY+MY:Gosub TVAL
```

where *X* and *Y* hold the MAP coordinates of the tile to be checked.

After TVAL has done its work, the icon at our intended destination is held in the variable *T*. We can now test this value directly inside our program. It's easiest to divide our tiles into a number of separate groups.

FLOOR tiles represent the tile numbers which can be walked across by the character.

WALL tiles are parts of buildings, or walls, which form a natural obstacle to the players movements.

CHARACTER tiles denote one or more characters which are controlled by the player.

MONSTER tiles - the baddies!

NPC tiles represent individual characters controlled by the computer. NPCs can be talked to, traded with, or even persuaded to join up with the party.

TRAP tiles are used to spring traps and trigger important events. They're usually assigned to the same images as the FLOOR or WALL tiles. So it's impossible to distinguish them from the background.

OBJECT tiles represent the various objects and items which can be picked up and used in the game.

If we are creating our maps with TOME, we can allot a separate identification number to each tile type with the TILE VALUER option. We can then immediately test the current tile using a line like:

```
V=TILE VAL(X,Y,0)
```

This gets the value of the tile at map coordinate X,Y. TOME allows us to define up to four different values for each tile. So we need to specify the list number as well. That's the reason for the final zero.

If we're not using TOME, we'll need to arrange it so that the tile numbers are completely separate. We can do this by simply lining up the tiles in numerical order of their type. So the first set of tiles represents the floor, the second the wall, the third the characters and so on. Just to make life easy for ourselves, we can define a few variables to hold the required numbers.

Here's a fragment from the program in Example 5.3.

```
SFLOOR=1 : LFLOOR=5
```

SFLOOR sets the number of the first tile to be used for the floor, *LFLOOR* holds the number of the last.

```
SBUILT=6 : FBUILT=9:Rem Tiles 6 through 9 are used for the buildings
SCHAR=10 : LCHAR=14:rem tiles 10 to 14 for the player characters
SMONST=15 : LMONST=17:rem tiles 15 to 17 make up the monsters
```

See how I've placed each set of tile values in sequence, one after the other. Anyway, once we've defined our ranges, we can test if there's a floor tile at the new character position using:

```
X=CX:Y=CY:Gosub TVAL
If T<=LFLOOR
  Then
    Gosub OK
  Else
    Gosub ILLEGAL
  Endif
```

So the final test looks like this:

```
X=CX:Y=CY:Gosub TVAL
If CX>=0 and CX<MPX and CY>=0 and CY<MPY and T<=LFLOOR
    Then
      Gosub OK
    Else
      Gosub ILLEGAL
    Endif
```

In Example 5.3, I've reorganised the system a little to make it faster. Since we are moving in only a single direction at a time, there's no need to test all the directions at once. But hopefully, you should recognise the principle immediately.

Displaying the next section of the screen

This can be accomplished either with the TDO routine from Example 5.2, or with an advanced scrolling system like the ones I'll be showing you in Chapter 7.

TDO redraws the entire screen whenever the player crosses over one of the edges. It may be crude and slow, but in many cases, it will be all you need. After all, we don't judge RPGs by the smoothness of their scrolling effects, do we? The quality of the puzzles is much more important! For the present, we can display the screen with simply:

```
Gosub TDO
```

Redrawing character

Example 5.3 contains a small CHARMOVE routine for this purpose. CHARMOVE takes the coordinates of our character from CX,CY and redraws the appropriate tile numbers at the new screen location. A record is kept of the tile we are replacing. When the character moves, we can use this value to restore the background to its previous state.

Repositioning character at edge

This is slightly fiddly, as we need to remove the character from the current position and replace the background underneath. In my example program, I've split this routine into two parts:

CHARCLEAR removes the character tile from the present location, and returns the background area to its original condition.

CHARDRAW replaces the character tile at the new position in the map. Since this map is about to be completely redrawn by TDO, I've haven't bothered to replace it on the screen with PASTE icon. But if you're using an alternative drawing system, you can easily do this with a line such as:

```
Paste Icon TX+CX*TWIDTH,TY+CY*THEIGHT,SCHAR
```

Well, that's it! Let's have a look at the final program.

Example 5.3: Moving through a map!

```
Close Workbench :Close Editor
Rem Create a temporary message screen
Screen Open 1,320,20,2,Lowres
Curs Off : Cls 0 : Centre "<BUILDING MEMORY BANKS>"
Rem Load some sprite images off the Extras disc
If Length(1)=0
  Load "EXTRAS:sprite_600/aliens/monster2.abk"
End If
Screen 0:Rem do graphics on screen 0
Flash Off
Rem Fade screen to black
Fade 4:Wait 60
Curs Off : Cls 0
Hide On : Rem Turn off mouse
Rem Start up double buffering
Double Buffer
Rem Only use values of 16 and 32 for twidth and theight
TWIDTH=16 : THEIGHT=16
Rem define tile ranges
SFLOOR=1 : LFLOOR=5
```

```
SBUILT=6 : FBUILT=9
SCHAR=10 : LCHAR=14
SMONST=15 : LMONST=17
Rem Make some icons
MAKE_ICONS[TWIDTH,THEIGHT,0]
Rem This next bits for the MOUSE section
Rem Gosub MAKE_CONTROL
Rem Make some tiles for our characters
Rem The tiles I'm using are vaguely reminiscent of PACMAN
Rem But I wasn't quite up to generating something more human shaped
Rem Anyway, I've got an excuse. I was reading WATCHMEN at the time
MAKE_SMILEY[10,1]
Rem Make some monsters. These are taken straight off the SPRITE_600
Rem files on the Extras disc and converted into icons in my program
MAKE_MONSTERS[15]
Rem The next lines for my combat sequence (see later)
Rem MAKE_SMILEY[18,2]
Rem load the map
Gosub MAP_INIT
Rem set position of viewing area on map
TX=80 : TY=32 : BX=240 : BY=160 : Gosub TVIEW
Rem Position ten monsters at random locations
For B=0 To 10
  L:
  X=Rnd(MPX-1) : Y=Rnd(MPY-1)
  Gosub TVAL:Rem Check for a legal position
  Rem If the position is already occupied by a BUILDING or an alien
  Rem Jump back to L and get a new one
  If T>LFLOOR Then Goto L
  T=SMONST+Rnd(2):rem choose monster
  Rem Plot monster on map
  Gosub TPLOT
Next B
Rem Set initial character position
L2:
X=Rnd(SWIDTH-1) : Y=Rnd(SHEIGHT-1)
Rem Search for an unoccupied space
Gosub TVAL
If T>LFLOOR Then Goto L2
Rem Save background tile underneath character
OLDT=T : OLDCX=X : OLDCY=Y
Rem Plot character in map
T=SCHAR : Gosub TPLOT
Rem Set character coordinates in CX,CY
Rem These are measured in units of a single tile
Rem from the top left corner of the CURRENT screen window
CX=X : CY=Y
Rem Draw first section of the map
Rem MX,MY are the tile coordinates of the area of the map
Rem which will be displayed
MX=0 : MY=0 : Gosub TDO : Rem Map Do MX,MY for TOME
Ink 1 : Box STX-1,STY-1 To SBX+1,SBY+1
Pen 2 : Paper 0
F=0
Rem Fade back sprite colours
Fade 10 To -1
Wait 150
Screen Close 1
Rem Example of new keyboard speed for mouse section.
Rem Key Speed 10,10
```

```
Rem Main loop
Do Rem Call up joystick control routine
  Gosub CONTROL_CHAR
  Rem Print current character coordinates
  Autoback 1
  Locate 0,0 : Print "Character is at  :-";MX+CX;" ";MY+CY;"  ";
  Autoback 0
Loop
CONTROL_CHAR:
  Rem Control character
  Rem save last position
  OX=MX : OY=MY
  Rem Read each direction in turn
  Rem Left
  If Jleft(1)
    Rem Get tile number to the left of our character
    X=CX-1+MX : Y=CY+MY : Gosub TVAL : QT=T
    Rem If it's legal, continue with the test
    If T<=LFLOOR
      Rem Check for a legal coordinate value
      Rem and see if the character is at the left edge
      If MX>0 and CX=0
        Rem remove the character icon from its old position
        Gosub CHARCLEAR
        Rem Move the drawing point one place to the left
        Add MX,-1
        Rem CX=SWIDTH-1 Remove to insert the character at RIGHT
        Rem Replace character at new location
        Gosub CHARDRAW
        Rem Redraw new area of map
        Gosub TDO
      End If
      Rem If move is legal and within the current screen window
      Rem The OX=MX bit checks that the character hasn't already
      Rem been moved
      If CX>0 and OX=MX
        Rem Reduce Character coordinate in CX
        Dec CX
        Rem Move character
        Gosub CHARMOVE
      End If
    End If
  End If
  Rem Now repeat for a RIGHT movement
  If Jright(1)
    T=SCHAR : X=CX+1+MX : Y=CY+MY : Gosub TVAL
    If T<=LFLOOR
      If MX<MPX-SWIDTH and CX=SWIDTH-1
        Gosub CHARCLEAR
        Add MX,1 : Rem CX=0
        Gosub CHARDRAW
        Gosub TDO
      End If
      If CX<SWIDTH-1 and OX=MX
        Inc CX : Gosub CHARMOVE
      End If
   End If
  End If
  Rem Handle a Downward move
  If Jdown(1)
```

```
    T=SCHAR : X=CX+MX : Y=CY+MY+1 : Gosub TVAL
    If T<=LFLOOR
      If MY<MPY-SHEIGHT and CY=SHEIGHT-1
        Gosub CHARCLEAR
        Add MY,1 : Rem CY=0
        Gosub CHARDRAW
        Gosub TDO
      End If
      If CY<SHEIGHT-1 and OY=MY
        Inc CY : Gosub CHARMOVE
      End If
    End If
  End If
  Rem Deal with an upward move
  If Jup(1)
    T=SCHAR : X=CX+MX : Y=CY+MY-1 : Gosub TVAL
    If T<=LFLOOR
      If MY>0 and CY=0
        Gosub CHARCLEAR
        Add MY,-1 : Rem CY=SHEIGHT-1
        Gosub CHARDRAW
        Gosub TDO
      End If
      If CY>0 and OY=MY
        Dec CY : Gosub CHARMOVE
      End If
    End If
  End If
  MK=0
Return
Rem Move our character
Rem OLDCX,OLDCY holds the coordinates of the last recorded location
Rem OLDT contains the tile underneath the character
Rem CX and CY are the new coordinates of the character
Rem measure relative from the top left of the screen window
CHARMOVE:
X=OLDCX+MX : Y=OLDCY+MY : T=OLDT : Gosub TPLOT
Rem Redraw background tile at old position with PASTE ICON
Paste Icon TX+OLDCX*TWIDTH,TY+OLDCY*THEIGHT,T
Wait Vbl
Rem Save tile underneath character
X=CX+MX : Y=CY+MY : Gosub TVAL : OLDT=T
Rem Store character in map
T=SCHAR : Gosub TPLOT
OLDCX=CX : OLDCY=CY
Rem Redraw character icon on screen
Paste Icon TX+CX*TWIDTH,TY+CY*THEIGHT,SCHAR
Wait Vbl
Rem Copy new image into both logical and physical screens
Screen Swap : Wait Vbl
Screen Copy Physic(0) To Logic(0)
Return
Rem Draws a character in the map without displaying it
CHARDRAW:
X=CX+MX : Y=CY+MY : Gosub TVAL : OLDT=T
T=SCHAR : Gosub TPLOT
OLDCX=CX : OLDCY=CY
Return
Rem Restores icon underneath character to original tile number
CHARCLEAR:
```

```
X=OLDCX+MX : Y=OLDCY+MY : T=OLDT : Gosub TPLOT
Return
Rem Makes a few tiles for my character
Rem SI is the size from 1 to 4
Rem S is the first tile number to be defined
Procedure MAKE_SMILEY[SI,S]
  For G=0 To 4
    Ink G+1 : Circle S*7,S*7,S*7 : Paint S*8,S*8
    Ink 0 : Ellipse S*4,S*5,S*1,S*2 : Paint S*4,S*5
    Ellipse S*10,S*5,S,S*2 : Paint S*10,S*5
    Ellipse S*7,S*10,S*5,S*3 : Ellipse S*7,S*9,S*4,S : Paint S*11,S*10
    Ink G+1 : Bar S*3,S*7 To S*13,S*9
    Get Icon G+SI,0,0 To S*16-1,S*16-1
    Cls 0,0,0 To S*16,S*16
  Next G
End Proc
Rem Make monsters
Rem S is the first tile number to be defined
Procedure MAKE_MONSTERS[S]
  Rem Read number of monsters
  Read MN
  Rem Copy monsters from sprite bank to icon bank
  For M=0 To MN-1
    Read B
    Hot Spot B,0,0
    Paste Bob 0,0,B
    Wait Vbl
    Get Icon S+M,0,5 To 15,20
    Cls 0,0,0 To 16,16
  Next M
  Cls 0,0,0 To 32,32
  Data 3,1,8,14
End Proc
Rem Add MAKE_ICONS, TINY_TAME, TVIEW. TDO, TVAC, TPLOT and MAP_INT
Rem Procedures from Example 5.2
```

5.7.3 Using the mouse

The Amiga's mouse pointer provides an extremely effective mechanism for controlling the activities of our party. AMOS Basic includes a powerful series of ZONE commands, which makes it easy to implement complicated dialogue boxes using the mouse pointer. These screen zones can be used to control everything from character generation to combat. I mentioned these zones in Chapter 4. But to jog your memory, here's a reminder of the general technique.

❑ Reserve some memory for your zones using RESERVE ZONE

❑ Initialise the zone system with RESET ZONE

❑ Define each button with SET ZONE. The format is:

`SET ZONE n,tx,ty TO bx,by`

This defines a rectangular button from *tx,ty* to *bx,by* and assigns it to zone number *n*

❑ Find the zone at the current mouse coordinates with:

`Z=Mouse Zone`

If the mouse isn't over a button, a value of zero will be returned. Otherwise *Z* will be loaded immediately with the zone number.

Here's a new version of the control system for use with Example 5.3. In order to use it, you'll need to add the following routines at the end of the program:

```
MAKE_CONTROL:
Rem allocate some space for your screen zones
Reserve Zone 4 : Reset Zone
Rem draw cursor arrows
DY=1
Rem Up and Down arrows
Repeat
  D=DY : X=264 : Y=16+(DY-1)*32+150 : W=16 : H=16 : ZN=DY
  Gosub MAKE_ARROW
  Inc DY
Until DY>2
DX=3
Rem Left and right arrows
Repeat
  D=DX : X=(DX-2)*32+216 : Y=182 : W=16 : H=16 : ZN=DX
  Gosub MAKE_ARROW
  Inc DX
Until DX>4
Rem Define some useful constants
AUP=1 : ADOWN=2 : ALEFT=3 : ARIGHT=4
Return
MAKE_ARROW:
Rem Requires the following variables to be set
Rem d=direction from 1 to 4
Rem ZN=Zone number
Rem X,Y=Starting coordinates of arrow
Rem W,H=Width and Height of arrow head
Set Zone ZN,X,Y To X+W,Y+H
Ink 5
On D Goto IUP,IDOWN,ILEFT,IRIGHT
ILEFT:
Polygon X,Y+H/2 To X+W/2,Y To X+W/2,Y+H To X,Y+H/2
Bar X+W/2,Y+H/4 To X+W,Y+(3*H)/4
Return
IRIGHT:
Polygon X+W,Y+H/2 To X+W/2,Y To X+W/2,Y+H To X+W,Y+H/2
Bar X,Y+H/4 To X+W/2,Y+(3*H)/4
Return
IUP:
Polygon X+W/2,Y To X+W,Y+H/2 To X,Y+H/2 To X+W/2,Y
Bar X+W/4,Y+H/2 To X+(3*W)/4,Y+H
Return
IDOWN:
Polygon X+W/2,Y+H To X+W,Y+H/2 To X,Y+H/2 To X+W/2,Y+H
Bar X+W/4,Y To X+(3*W)/4,Y+H/2
Return
```

You can now remove the REM statement around the line:

```
Gosub MAKE_CONTROL
```

reactivate the mouse by removing HIDE ON and replace CONTROL_CHAR with:

```
CONTROL_CHAR2:
Rem Just for a bit of extra mayhem, I've also included
Rem The ability to read the cursor arrows from the keyboard
Rem
```

```
Rem Clear command string
K$=""
OX=MX : OY=MY
Rem Read mouse zone
D=Mouse Zone
Rem Read keyboard
K$=Inkey$
Rem Add any keyboard commands here!
Rem read status of mouse keys
MK=Mouse Key
Rem Move through map
Rem ALEFT,ARIGHT,AUP,ADOWN are the zone numbers of
Rem the arrow buttons
If(D=ALEFT and MK=1) or(K$=Cleft$) or Jleft(1)
  X=CX-1+MX : Y=CY+MY : Gosub TVAL : QT=T
  If T<=LFLOOR
    If MX>0 and CX=0
      Gosub CHARCLEAR
      Add MX,-1 : Rem CX=SWIDTH-1
      Gosub CHARDRAW
      Gosub TDO
    End If
    If CX>0 and OX=MX
      Dec CX : Gosub CHARMOVE
    End If
  End If
End If
If(D=ARIGHT and MK=1) or(K$=Cright$) or Jright(1)
  T=SCHAR : X=CX+1+MX : Y=CY+MY : Gosub TVAL
  If T<=LFLOOR
    If MX<MPX-SWIDTH and CX=SWIDTH-1
      Gosub CHARCLEAR
      Add MX,1 : Rem CX=0
      Gosub CHARDRAW
      Gosub TDO
    End If
    If CX<SWIDTH-1 and OX=MX
      Inc CX : Gosub CHARMOVE
    End If
  End If
End If
If(D=ADOWN and MK=1) or(K$=Cdown$) or Jdown(1)
  T=SCHAR : X=CX+MX : Y=CY+MY+1 : Gosub TVAL
  If T<=LFLOOR
    If MY<MPY-SHEIGHT and CY=SHEIGHT-1
      Gosub CHARCLEAR
      Add MY,1 : Rem CY=0
      Gosub CHARDRAW
      Gosub TDO
    End If
  If CY<SHEIGHT-1 and OY=MY
      Inc CY : Gosub CHARMOVE
  End If
  End If
End If
If(D=AUP and MK=1) or(K$=Cup$) or Jup(1)
  T=SCHAR : X=CX+MX : Y=CY+MY-1 : Gosub TVAL
  If T<=LFLOOR
    If MY>0 and CY=0
      Gosub CHARCLEAR
```

```
      Add MY,-1 : Rem CY=SHEIGHT-1
      Gosub CHARDRAW
      Gosub TDO
    End If
    If CY>0 and OY=MY
      Dec CY : Gosub CHARMOVE
    End If
  End If
End If
MK=0
Return
```

5.8 Checking for an Encounter

As our party marches through the game world, it will be continually faced with a variety of obstacles. These can be anything from a terrifying alien monster to a locked door.

In order to include these events in our games, we'll need to be able to store the position and types of these obstacles directly in our map data. As I've mentioned previously, this can be accomplished by creating special tiles for our monsters, objects, doors and traps. We can now check for the required event by just searching the area immediately around our party for the appropriate tile values.

5.8.1 A simple scanner

Supposing our characters were at map coordinates PX,PY. We could perform our search using a routine like:

```
For X=PX-1 To PX+1
  For Y=PY-1 To PY+1
  Rem Check that X and Y are legal map coordinates
  If X>=0 and X<MPX and Y>=0 and Y<MPY
    Rem Get tile number
    Gosub TVAL
    Rem Check for a monster tile
    If T>=SMONST and T<=LMONST
      MT=T-MONST
      On MT+1 Gosub MONSTER1,MONSTER2,MONSTER3
    Endif
    Rem Check for a door tile
    If T>=SDOOR and T<=LDOOR
      DT=T-SDOOR
      On DT+1 Gosub DOOR1,DOOR2,DOOR3
    Endif
    Rem check for a trap tile
    If T>=STRAP and T<=LTRAP
      TT=TT-STRAP
      On TT+1 Gosub TRAP1,TRAP2,TRAP3
    Endif
  Endif
  Next Y
Next X
```

This routine scans the area around the party one tile at a time, and checks it for a monster, a door or a trap. MONSTER1, MONSTER2 and so on are assumed to be previously defined routines in our program.

MPX and *MPY* hold the width and height of the map in tiles.

Figure 5.2 contains a pictorial representation of the testing procedure. The tiles are tested in the order 1, 2, 3... *c* is the position of the character.

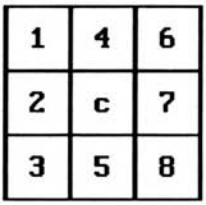

Figure 5.2: Scanning the surroundings

5.8.2 Advanced scanners

In reality, the scanning routine would need to be considerably more complicated. Here's a real example, which will be used in my demo game at the end of this chapter.

```
SCAN:
Rem Scans the area surrounding the adventurer
Rem Gets the starting point of the current map data from bank six
S6=Start(6)+4
Rem Sets the distance to be checked on each side of the character
DX=1 : DY=1
Rem Sets the distance to be checked on each side
SCX=-DX
Rem Combat is a variable which will be either TRUE(-1) or FALSE(0)
Rem It's used to tell the rest of the program that a monster has
Rem Been detected
COMBAT=False
Rem MONST holds the number of monsters
MONST=0
Rem See how I've substituted a REPEAT UNTIL loop for my FOR..NEXT
Rem There's no spectacular reason for this. But some versions of
Rem the AMOS compiler can't handle GOSUB's within FOR..NEXT loops
Rem
Repeat
  Rem Sets the distance to be checked up and down
  SCY=-DY
  Repeat
    Rem Find map coordinates of character
    PX=CX+MX+SCX : PY=CY+MY+SCY : MP=PY*MPX+PX
    Rem If PX>=0 and PX<MPX and PY>=0 and PY<MPY
      If PX>=MX and PX<MX+SWIDTH and PY>=MY and PY<MY+SHEIGHT
      Rem This is the guts from TVAL
        T=Peek(S6+MP)+1
        Rem Check for a monster
        If T>=SMONST and T<=LMONST
          MT=T-SMONST : COMBAT=True
          Rem Jump to a routine to handle the monster
          On MT+1 Gosub BLOB,SNAIL,BEHOLD
          Rem Load the TYPE and NUMBER of the monster in
          Rem MROUND array
          MROUND(MONST,0)=MT : MROUND(MONST,1)=MN
          Inc MONST:Rem Next monster
        End If
```

```
        Rem Handle objects (in this case they are AMULETS)
        If T>=SAMULET and T<=LAMULET
          Rem Get amulet type
          AN=T-SAMULET
          Rem Amulets have magical healing properties (Why not!)
          HEALTH(1)=STA(1)+AN+1
          Bell 9
          Rem Display each amulet in a fancy box on the screen
          Autoback 1
          Ink 2
          Box AMULETS*20,188 To(AMULETS+1)*20-1,210
          Paste Icon AMULETS*20+2,192,T
          Inc AMULETS
          Wait Vbl
          Autoback 0
          Rem Remove amulet from the map
          T=SFLOOR : X=PX : Y=PY : Gosub TPLOT
          Rem And redraw the map
          Gosub TDO
        End If
      End If
      Rem End If
      Inc SCY
    Until SCY>DY
    Inc SCX
Until SCX>DX
Return
```

Hopefully, much of this should be pretty familiar. But you're probably wondering a little about the MROUND array.

5.8.3 Generating a monster!

Let's start from first principles. In an arcade game, each monster is basically just a blob on the screen to be destroyed without mercy. But in a role playing game, our monsters are much more realistic. They have their own individual personality and therefore require their own unique list of attribute values. Like the character attributes, these numbers can be generated in advance, or can be produced directly when they are needed in our program.

Thankfully, monsters are much simpler than characters. So they don't need the full complement of attributes such as intelligence or sex. However, since we'll be using hundreds of monsters in our game, each attribute needs to be crammed into the minimum possible space. The best way of achieving this, is to pack them into a single string variable using the CHR$ function. CHR$ (pronounced cars) is one of these lovely arcane instructions which everybody uses, but nobody talks about! If you've never encountered it before, here's a quick explanation.

C$=CHR$(n)

This converts a number from 0 to 255 in *n* into the equivalent text ASCII character. ASCII stands for American Standard Code for Information Interchange. It's really just a standard way of storing letters in a computer.

Examples:

```
print chr$(32)
```

prints a space.

```
print chr$(49)
```

1

```
print chr$(97)
```
a

```
Rem This was the first program I ever typed in AMOS Basic
Rem in June 1989!
Rem It just prints out all the characters on the screen
For C=32 to 255
  Print Chr$(C);
Next C
```

This prints out all the available characters on the Amiga's screen. The values from 0 to 31 are normally used to control the cursor, and activate the various writing modes. So I haven't bothered to include them.

As you may have guessed, there's also an ASC() function as well. This returns the ASCII value associated with each letter or character.

```
print ASC(" ")
```
32

```
print ASC("a")
```
97

```
print asc("1")
```
49

Back to work! These commands provide the ideal way of packing all our statistics into a single string variable:

```
S$=Chr$(STR)+Chr$(SPD)+Chr$(STA)+Chr$(HEALTH)+Chr$(DAM)
```

We can now define a string array for each monster type, and load our attribute values straight into the string with:

```
S$=Chr$(STR)+Chr$(SPD)+Chr$(STA)+Chr$(HEALTH)+Chr$(DAM)
BEHOLDER$(MN)=S$
```

Ok, so we've defined our attributes, saved them into our string array, and positioned our monsters. What's left? Well, we still need to be able to discover which particular monster has been found at the present map position. Remember that in a real game, there could be hundreds, or even thousands of each of the various monster types. Unless we can accurately identify the one we are currently facing, we'll be completely unable to get at its attributes.

5.8.4 Identifying a monster

We'll begin by defining an array to hold the map position of each monster in our game.

```
Dim MPOS(mtypes,mnumber)
```

mtypes is a constant which contains the maximum number of types in the RPG. Similarly *mnumber* sets the maximum number of monsters which can be assigned to any one species.

While we're at it, it's logical to store a running count of all the monsters assigned to a particular species. These can be placed in a separate array, and will be invaluable when we come to searching through the final position list.

```
Dim MCOUNT(mtypes)
```

At the start of our program, we can now store the position of each monster into our MPOS

array. We could hold this location as a couple of X,Y coordinates, but it's faster to combine them into a single large value.

position=(Y coordinate * number of columns)+ X coordinate

This value represents the distance from the start of the tile bank to the present map position. Since my scanner routine needs to read the tile value directly from the memory bank, the current monster position is already available in this format. It's been placed in the variable *MP* by:

```
MP=PY*MPX+P
```

So we don't have to do any extra calculations.

The final search program looks like this:

```
Rem Finds the number of the current monster
Rem inputs MN=monster type
Rem MP=Monster position
MFIND:
Rem Get the number of monsters assigned to the current type
C=MCOUNT(MT)
Rem And search through the list in MPOS until we find the monsters
Rem Or run out of positions
MN=-1
Repeat
  Inc MN
Until MPOS(MT,MN)=MP or MN>C
Return
```

The same ideas can also be applied to all the other articles in the adventure, such as objects, traps or NPCs. All we need to do, is replace MPOS with the equivalent position array.

Here's a complete monster handler for you to examine as well.

```
SNAIL:
Rem This is used to handle a Giant snail
Rem It would be called by the SCAN routine in section 5.8.2
Rem Snails are neat animals. They are very large, and extremely
Rem SLOW. Rather like me!
Rem
Rem find the snail number at the current position
Gosub MFIND
Rem If it's not found, return with a error
If MN>C Then Return
Rem Get statistics for this snail
S$=SNAIL$(MN)
Rem If they don't exist, then create some more
If S$=""
  Rem snails are slow, tough, but pack a mean punch!
  STR=Rnd(20)+15 : SPD=Rnd(2)+2 : STA=Rnd(20)+30 : HEALTH=STA
  DAM=10+Rnd(10)
  Rem pack attributes into a string.
  S$=Chr$(STR)+Chr$(SPD)+Chr$(STA)+Chr$(HEALTH)+Chr$(DAM)
  SNAIL$(MN)=S$
End If
Rem Now call a separate routine to load the statistics into a
Rem series of temporary arrays used by the COMBAT system.
Rem see later
Gosub LSTATS
Return
```

5.9 Multiple Characters

In order to capture the feel of the original multi-player RPGs, it's common practice for the computer versions to provide the player with a large adventuring party consisting of a number of individual characters. But the character in an RPG needs to be a believable extension of the player's personality. Since people are only able to relate directly to a single character at a time, multiple characters inevitably remove much of the role playing out of the RPG. So why do most games still bother to include them?

The answer is that, used properly, multiple characters can add an intriguing element of strategy to our games. The player now has the additional responsibility of selecting the mix of character types in the party. Each character type has its own combination of strengths and weaknesses, and the player needs to manage these abilities to the party's best advantage. Furthermore, by choosing a different arrangement of characters, the same game can be played over and over again.

5.9.1 The leading character

If we were to ask the player to control the movements of four or five characters on the screen independently, the action of the game would naturally be reduced to a crawl. Because of this, it is standard practice to nominate one character from the group as the leader.

If the party starts off with a single character, the choice should be taken out of the player's hands. This will allow us to add a little autonomy to the rest of the characters. In real life, not all the people we meet in our travels are either honest or trustworthy. The addition of realistic characters with their own personality can enhance the atmosphere of even the simplest game.

Games which display a view of the party from above can avoid a great deal of unnecessary complication by only representing the leader on the standard game map. A three dimensional display should also be assumed to show the current surrounding entirely from the view point of the current leader.

5.9.2 Individual Control

Individual control only becomes important during combat. Each character in the group needs to be given the specific orders for the attack. These orders need to be input at high speed to keep the action of the game running smoothly.

Some games utilise a separate screen for this purpose, and allow each player to be moved around independently. Experience has shown that the combat element of these games is often dangerously unwieldy. If we really want to use individual control, we would be better to allow the input of lists of orders to the character in advance. This would bring the RPG much closer to a genuine war-game.

5.10 Combat

It is the combat elements of a role playing game which set it apart from the run of the mill adventure. When we are writing our games, we should therefore take particular care to make the fighting as fast and as furious as possible.

5.10.1 Melee rounds

All the combat in the game is split into a number of separate melee rounds. These simulate the results of a single exchange of blows between two opposing fighters. Combat is performed in the following stages:

```
Find monsters and set up attack mode
Repeat
    Repeat for each character
        choose weapon
        choose an opponent
       check whether weapon hits the opponent
        if hit is scored
        then
            determine damage
            check effect of damage
        endif
        Let the opponents fight back
        check whether a weapon hits the character
        if hit is scored
        then
           determine damage
            check effect of damage
        endif
    Until last character
Until combat ends
```

It's necessary to perform these steps for every character involved in the fighting. The battle will continue until all members of one side are dead, or one group of combatants retreats.

5.10.2 Types of combat

Just to make life more complicated, there are also several distinct types of combat which can occur in an RPG:

Close Combat

This is old-fashioned hand-to-hand fighting, using swords and shields. All the action takes place in a limited area, and the characters have to block or deflect each individual blow as it comes.

Ranged Combat

Ranged combat applies to gun-fights, where the combat involves an exchange of shots. In order to survive, the characters have to dive immediately for the nearest cover, and then carefully loose off their shots at selected targets. They'll also need to keep track of the amount of ammunition!

Magical Combat

Magical combat is basically a variation of the Ranged system, but it's complicated by the various magical elements. Although spells like Fire Ball are just magical variations of modern weapons, others such as Invisibility or Befuddle, can have a dramatic and unpredictable effect on the entire course of the battle.

In addition to the combat systems, there are two ways of controlling our characters:

Real-Time

This simulates the fighting as it's happening. A single second in the real world corresponds to one second in the game world. Real-time combat is perfect if we're controlling just a single character. But if the party is larger than about two, the events in the game move far too quickly. The player is overwhelmed by the need to enter a flood of orders, and the party stands around aimlessly while the monsters attack!

Game-Time

The only way of controlling big parties is to use some sort of advance ordering system. The player gives a series of commands to each party member in turn, and then hits a single key to start the fight sequence. The results of the melee round are then calculated by the computer, and are presented neatly on the screen. The player is given the option of continuing with the round, or pulling the party out, terribly fast.

I'll now break down each aspect of the combat system into AMOS routines.

5.10.3 Starting out

The first thing we need to do is find the monsters and load their combat statistics into some temporary arrays. Although the string based system in Section 5.8.3 uses very little memory, it would be impossibly complicated for our actual combat sequence.

Finding the monsters is easy, as the SCAN routine has already loaded the number and type of our opponents into the MROUND array. Remember:

MROUND(N,0) holds the TYPE of OPPONENT number N
MROUND(N,1) stores the monster's individual identification number

We can use these numbers to grab the stats of our monster into the appropriate attribute arrays.

```
LSTATS:
Rem MSTR,MSPD,MSTA,MHEALTH,and MDAM are arrays which
Rem are assumed to have been defined earlier in our program
S=0
Rem Load STRENGTH of monster into MSTR
Rem VARPTR finds the Address of string S$ in the Amiga's memory
Rem Peek reads the contents!
ST=Peek(Varptr(S$)+S) : MSTR(MONST)=ST : Inc S
Rem Load SPEED of monster into MSPD
ST=Peek(Varptr(S$)+S) : MSPD(MONST)=ST : Inc S
Rem Load stamina of monster into MSTA
ST=Peek(Varptr(S$)+S) : MSTA(MONST)=ST : Inc S
Rem load HEALTH of monster into MHEALTH
ST=Peek(Varptr(S$)+S) : MHEALTH(MONST)=ST : Inc S
Rem Load DAMAGE of monster in MDAM
Rem DAMAGE effects the amount of the injury which will be
Rem inflicted when the monster hits a character (see later)
ST=Peek(Varptr(S$)+S) : MDAM(MONST)=ST
Return
```

We first encountered this routine in Section 5.8.4, where it was called up as part of the SNAIL handler. This was the module which identified our original monster and loaded its statistics from memory.

Further examples can be found in the BEHOLD and SNAIL routines in the demo game on the Companion Disk to this book. Don't get confused with the PEEK and VARPTR stuff incidentally. These are just a fast way of getting the attribute number *S* out the statistics string *S$*. They're equivalent to the line:

```
ST=Asc(Mid$(S$,S,1))
```

Once we've loaded our monsters, we are ready to take up our arms, and FIGHT!

5.10.4 Weapons

Unless we are intending to attack the monsters with our bare hands, we'll need to devise some sort of weapons system.

For the purposes of my discussion a weapon can be taken as anything which is capable of inflicting damage. This includes natural weapons like claws, teeth and fists, as well as traditional items such as swords and axes.

The effect of a weapon is determined by two main factors:

Damage

This sets the average amount of punishment which will be inflicted if the weapon successfully connects with an opponent. The damage factor varies considerably depending on the type of weapon and the statistics of the character wielding it. In the case of close combat, strong characters will naturally do rather more damage than weaker ones.

Hit Chance

This determines the probability of the weapon hitting the intended target. As before, this depends on a number of factors including the weapon type and the relative speeds of the attacker and the defender.

The precise method of choosing these values for a particular weapon is clearly a matter of debate. Each Real RPG system has its own unique combat routines. So there's obviously plenty of scope for experiment. Personally, I prefer to keep things simple. So I record the Damage factors for each weapon in a single WEAPON array.

```
Dim WEAPON(10):rem space for ten weapons
```

I then base the Hit Chance entirely on the relative abilities of the combatants. This approach works fairly well, but if you'd like something a bit more realistic, you can easily expand it if required. You could, for instance, store a complete list of additional statistics for each weapon.

Here's an example:

```
Dim WEAPON(10,4):Rem reserve space for ten weapons and five attributes
```

We could now define our attributes like so:

WEAPON(N,0) holds the basic chance of the weapon hitting.
WEAPON(N,1) stores the DAMAGE factor
WEAPON(N,2) specifies the type (RANGED,EDGED,IMPACT,SPEAR..etc)
WEAPON(N,3) contains the size
WEAPON(N,4) indicates the weight

These attributes could be included as part of our combat system. If you were feeling really eager, you could even define a separate routine to handle all the types of weapons. This could be called at the start of our combat module using:

```
WT=WEAPON(N,3)
On WT+1 Gosub RANGED,EDGED,IMPACT,SPEAR
```

5.10.5 Selecting the weapon

Each character in the current party will have a number of different weapons in his or her possession. As I mentioned in Section 5.5.3, the number of this weapon will be stored in the SACK and OTYPE arrays.

SACK(character,objects) holds the list of items carried by each character.
OTYPE(item_no,attributes) stores the attributes for each item in the game.

In addition to these arrays, it's sensible to keep an independent count of the amount of damage done by the currently selected weapon. This can be stored in a DAMAGE array like so:

```
Dim DAMAGE(no_of_characters)
```

This allows us to get an instant read-out of the Damage factor for each member of the party. I've also used it extensively for my monsters, as it's clearly pointless to hold a complete list of objects for each monster.

Some games allow the characters to hold a weapon in each hand, ready for immediate use. We can include this in the present system by simply tacking an extra dimension to the previous DAMAGE array.

```
Dim DAMAGE((no_of_characters,no_of_active_weapons)
```

Note that I haven't concerned myself about how these Damage factors are actually calculated. This depends entirely on the weapons system you are using in your game, and can be as simple, or as complicated as you like.

5.10.6 Selecting the opponent

The opponent can be selected automatically by our program, either randomly, or by taking account of the relative positions of the characters on the screen.

In my demo game I've assigned a simple number to each of the adversaries. The player can now select between the various enemies by pressing T (for Target), and entering a single digit from the keyboard.

5.10.7 Determining the success or failure of a blow

This is complicated by the fact that members of the party can attempt to dodge away from the blow. The likelihood of a successful parry will largely depend on the relative speed of the two combatants. If an agile character tries to block a clumsy opponent, the defence will probably succeed. Similarly, a fast blow against a slow defender stands a very good chance of hitting.

Real RPGs use a range of methods of calculating the combat results. So there are literally hundreds of ways implementing these techniques in a computer game. However, the easiest system is to work out the ratio of the two Speed values like so:

CHANCE=(SPEED of defending player)/(SPEED of attacker)

We could then check for the success of a parry with something like:

```
HCHANCE#=(PSPEED#/MSPEED#)*50
R=Rnd(100)
If R<HCHANCE#
  Gosub DAMAGE
Else
  Gosub PARRY
Endif
```

See how I'm using Real variables for these calculations. This is essential, as the standard AMOS variables are only capable of storing whole numbers. So a division such 10/7 gives 1 rather than 0.7. The 50 used in the HCHANCE# calculation was generated after experimentation. The higher the value, the greater the importance of speed in the chances of a dodge. This system represents the simplest option we can use. In a real program, it would normally be expanded slightly.

Here's a fragment from a real program:

```
If REAL_TIME : XT=(100-WT)/25 : Else XT=0 : End If
MO#=MSPD(MN) : PL#=SPD(1)+XT
HCHANCE#=Min(BASE,(PL#/MO#)*50)
R=Rnd(100)
```

The Min command sets the maximum chance of a successful hit to base. Usually this is set to around 95%. So there's still a small possibility of deflecting even the fastest blow. If the characters are completely outmatched, they may decide to concentrate all their efforts on defence, increasing the odds on a successful parry. I've simulated this in Example 5.4 by temporarily improving the character's strength:

```
FDODGE:
Rem Save the old speed
OSPD(1)=SPD(1)
Rem increase speed of a character
SPD(1)=SPD(1)+STR(1)/4
Rem Extra speed if fit and well
If HEALTH(1)>10 Then SPD(1)=SPD(1)+1
Rem Set up a variable to tell the combat routine of the dodge
DGE=1
Return
```

A similar idea can also be applied to handle the player's retreats.

5.10.8 Assessing the damage

The amount of damage inflicted on by a particular blow depends on the Damage factor of the current weapon and the Strength of the attacker. As with the Hit Chance, the significance of the strength factor in these calculations is up to you. The general equation is:

D=Rnd(DAMAGE)+STRENGTH/R

where *R* is a reduction factor which controls the minimum amount of damage. If *R* is set to 1, every time a character hits something, he will do Strength points of damage.

Unless the stamina of the opponent is extremely large, it will die after more than a couple of blows. This leads to short and murderously lethal combat rounds. What's more, we'll be using the same equation to assess the injuries inflicted on the player's character. So unless the monsters are very weak, the character will be quickly killed off. It's therefore sensible to increase *R* to a value such as 2 or 4. This limits the minimum amount of damage to a fraction of each character's strength, and improves the playability of the game no end.

Also note that the Damage factor needs to be chosen with equal care. If the weapons in your RPG are too powerful the player characters won't stand a chance. But if they are too weak each combat round will take half an hour to play out.

Once we've assessed the damage, we'll need to see if the character or monster has been seriously injured. Here a few of the more common approaches we can take:

- Subtract the damage from the character's health. If this becomes less than or equal to zero, then we can assume that the character has been killed off. Easy, and simple. A typical line would be:

```
D=Rnd(DAM(1))+STR(1)/4 : Add MHEALTH(MN),-D
```

  where *MHEALTH* holds the Health of the monster, and *DAM* and *STR* store the Damage and Strength attributes of the player characters.

- Divide the damage evenly between the attributes Stamina, Strength or Speed. Only kill off the character when a couple of attributes reach zero. This is avoids the ridiculous possibility of a dying character jumping from the ground and chopping a monster's head off with a single blow. On the other hand, it's also more complicated, since we now have to keep track of all the original attribute settings. It could be implemented using something like:

```
D=Rnd(DAM(1))+STR(1)/4
Add MSTR(MN),-D/3
Add MSTA(MN),-D/3
Add MSTR(MN),-D/3
```

The final element in a combat situation is the possibility of fumbles. Whenever several people are wielding weapons around, there's always the real danger that they will hit one of their allies by mistake. Many Real RPGs assign a specific die roll such as a double zero for this purpose. If the player rolls this value, something disastrous is assumed to have taken place, and further dice are rolled to determine the final outcome. This can vary between the character accidentally shooting himself in the foot, or decimating his entire party! The same technique can also be applied in one of your own games. All you need to do, is test for the possibility of a fumble, and then insert the appropriate IF..THEN statement to detect it. The result of the fumble is entirely up to you.

5.10.9 Armour

Another complication is raised by the possibilities of armour. The net effect of this will be to reduce the impact of a blow by a fixed amount, depending on the strength of the armour. This can be implemented using a simple offset which can be subtracted from the damage inflicted on the player.

Different types of armour will absorb different amounts of damage. This can be represented by an attribute such as the Defence factor. The higher it is, the better the armour. We could store this information in the arrays:

```
Dim ARMOUR(100)
```

Holds the Defence factor for each type of armour.

```
Dim WORN(10)
```

Stores the Defence factor of the armour worn by our characters.

```
Dim MWORN(10)
```

Similarly for our monsters.

Here's a typical system:

```
D=Rnd(DAM(1))+STR(1)/4
HIT=D-MWORN(MN)
Add MHEALTH(MN),-D
```

5.10.10 Ranged combat

Ranged combat is normally performed in real-time, so we can control our characters directly on the screen. The general principles of these systems can be seen from the following snatch of pseudo-code:

```
Enter target coordinates
Repeat
  check for a collision with a monster
  If collision
    Then
      Find monster
      Calculate damage
    endif
  check for a collision with a wall
  check distance from gun
  move missile
Until an object is hit or the missile exceeds the maximum range
```

Entering the target coordinates is easy, as we can simply create some cross-hairs, and position them over our map with the mouse or joystick.

Our missiles can be created using either sprites or Bobs, and can be moved using the standard AMAL commands. The only tricky bits, are checking for a collision, and testing the range. Since most of our objects only exist inside the map data, we can't use any of the normal collision detection commands. We'll therefore need some way of finding the tile number currently underneath the missile. Unfortunately, all AMAL commands are performed extremely fast. So if we tried to keep an accurate track of our missile as it moves on the screen, we'd be doomed to failure.

The solution, is to move our missile a small step at a time, using the SYNCHRO command I showed you in Chapter 3. We can now carefully check for a collision using TVAL, before the missile continues on its journey.

Remember:

SYNCHRO OFF deactivates the automatic AMAL movement system completely. **SYNCHRO** executes a single instalment of our AMAL commands in our Basic program.

In order to track our missile, we'll need to keep a continual record of its position on the screen. This can be found using the functions:

X BOB/Y BOB for blitter objects

or

X SPRITE/YSPRITE for sprites.

We can then convert these coordinates into the equivalent map position, and read the number of the appropriate tile straight from our map data.

Checking the range requires us to indulge in a little O-level maths. That's because we need the direct line distance from our character to the target. This can be generated using the following procedure:

- Measure the horizontal and vertical distances from the character to the target. If the character coordinates were in CX,CY and the target coordinates in TX,TY, the required distances would be:

```
XP#=TX-CX
YP#=TY-CY
```

As you can see, I'm loading the distances into Real variables rather than the standard integers. You'll see why in a few moments.

❑ Calculate the straight line distance from the player to the target using Pythagoras theorem.

```
R#=Sqr((XP#^2)+(YP#^2))
```

I appreciate that if you're not into Trigonometry this equation may look pretty horrible. But providing you remember to use Real variables in the calculation, you can use it directly, without having to understand the reasoning behind it. Note that SQR finds the square root of a real or decimal number. Ok?

An example is provided in the demo game on the companion disk.

5.11 Magic

Magic is an integral part of many role playing systems. The big problem of course, is that it is totally confined to the player's imagination. Like it or not, in the real world, magic simply doesn't work. So while we may know a little about the swords part of Swords and Sorcery, the sorcery bit is completely outside our actual experience.

The only way to successfully apply magic in a game is to carefully invent a logical and consistent system of rules. Although the laws of physics may be different in your game world, you should always try to keep within the bounds of common sense. The fact is, unless the players are provided with a convincing rationale which underlies your system, they will simply refuse to believe in it.

In practice, you will undoubtedly need to place real restrictions in the capabilities of your magic users. Otherwise there will be nothing to stop a powerful wizard zapping even your toughest monsters with just a casual wave of his hand.

There are several possible solutions to this difficulty.

❑ Assume that all spells need a material component of some sort which is used up after the spell has been cast. Really powerful spells might need especially rare items in order to work. An example of this technique can be seen in the excellent Ultima games. You could even use the same idea as the basis of the entire scenario.

❑ Make wizards start off physically weak and limit their initial abilities ruthlessly. As the characters increase in power, you can progressively confront them with tougher and tougher monsters.

❑ Keep an indicator of a character's current power rating, which can increase with experience. In order to cast a particular spell, the character needs to temporarily expend a certain amount of magical energy. By assigning the highest power requirements to the more effective spells, you can easily restrict the strongest magic to characters who have survived some way into your RPG. This is used in the vast majority of D&D type games, and works well.

❑ Assign a cryptic code to each spell. If a character doesn't know the code, then it is impossible to cast the particular spell. You can place these codes on scrolls at appropriate points in your game as a reward for solving a especially difficult puzzle. This is the approach used in FTL's Dungeon Master program.

It is perfectly possible to exploit any combination of these ideas in a single game. After all you're the boss of this world. So quite literally anything goes.

5.12 Conclusion

Creating a role playing game can be an highly enjoyable pastime in its own right. I've personally refereed several Real RPGs and I can assure you that they are terrific fun to organise. You can now capture the same feeling of satisfaction by writing your very own role playing game on the Amiga. With a little help from AMOS Basic, and a lot of hard work, you might well produce a game which is commercially viable. All you need are the ideas and the imagination. The rest is easy.

6

Simulations

6.1 What is a Simulation?

Simulations provide one of the most useful applications of computers in our modern world. The art of the simulation is to break down a complex physical system and mimic its behaviour with a series of mathematical equations. These equations can be subsequently solved with the help of a computer program, allowing us to make uncannily accurate predictions as to the outcome of events. So we can therefore test out our ideas directly inside the computer, without the risk of disastrous and expensive mistakes.

In reality, it's often essential to get things absolutely right on the very first attempt. When the scientists at NASA wanted to calculate the orbit for the Voyager space probe, they didn't just shoot off a couple of dozen ships in the hope that one would eventually hit the mark. That would have been sheer lunacy! Instead, they carefully simulated the orbit with a computer, and simply checked through all the possible combinations until they found one they especially liked.

This allowed them to harness Jupiter's enormous gravity field to neatly flick the probe from Jupiter to Saturn using practically no extra fuel. Without the ability to simulate the mission completely on the ground, the whole trip would have ended at Jupiter, and we'd all have been deprived of those breathtaking pictures of Saturn and its rings.

Of course, simulations are not always quite that effective. The Met office regularly runs some of the largest simulations in existence with the help of computers of literally awesome power. But the reliability of their weather forecasts is still a matter of heated debate!

Since even the most dedicated computer expert needs to goof-off occasionally, simulations have also formed the basis of a vast number of computer games. These are especially popular with trainee programmers, as many of the basic techniques are naturally acquired as part of their studies.

The earliest examples were games like Golf, Lunar lander and Kingdom. Originally these programs were unbelievably crude, and their graphics were almost non-existent. Despite this, they were great fun, and I still remember them with a great deal of fond nostalgia from my own days as a student.

Over recent years, the simulation game has evolved out of all recognition though. Golf has been transformed into games like World Class Leaderboard, and Lunar Lander has progressed from a crude simulation of a real spaceship, to the fantastic imaginary world of Elite.

The most surprising improvements however, have been in the original Kingdom game. This has provided the inspiration for a whole generation of classic programs, including well known favourites such as Sim City, Populous and Supremacy. AMOS is ideal for these economic simulations, as it includes superb graphical abilities along with the raw processing power to cope with even the most complicated mathematics. So let's journey onward, and discover how we can create one for ourselves!

6.2 An Introduction to Mathematical Modelling

At the heart of any simulation there is a series of linked equations known as a mathematical model. These equations concisely describe the underlying rules which govern the real system.

A mathematical model is rather like a cartoonist's sketch. When a cartoonist portrays a well-known political figure, the picture is created out of a simple sequence of lines and curves. These lines do not exist on the actual person of course! They are merely a tool used to bring out the essential personality of the character on a two-dimensional piece of paper.

The same is true of the various equations making up a mathematical model. Although these equations only exist inside the computer, their workings capture the essence of the real event we are trying to simulate. The job of the mathematical modeller is to decide precisely which *lines* should be drawn in the sketch, and which features should be exaggerated for effect.

There's no need to panic if you're not a mathematical genius! Neither am I! Most of the topics I will be discussing can be created with surprisingly basic maths. Otherwise, I'd never have been able to cope with them myself! Believe me, the mathematics in a simulation game are a far cry from the boring proofs you may have encountered at school!

There's actually a strong relationship between the abilities needed to complete a simulation and the skills required to create one from scratch. So although it will certainly help if you have a general flair for mathematics, the only real qualification is a genuine enthusiasm for simulation games themselves. Providing you're willing to persevere, everything else can be easily picked up as you go along.

6.2.1 The game world

The equations in the mathematical model form an imaginary game world which provides the backdrop for all the events in our game. In a commercial simulation, there will hopefully be a some direct correspondence between the events in the real world and those in the simulation. If there isn't, any results we get from the simulation will be meaningless. (Maybe we should tell the Met Office!)

In a game however, this link is purely arbitrary, and it's quite possible to simulate situations which would appear totally ridiculous in reality.

Take the hyper-space sequences found in games like Elite. Ask a respectable physicist about the possibility of hyper-drive, and you will be greeted with hoots of derision. Alas they'll say, according to the best modern theories, it's totally impossible to travel faster than the speed of light. They'll then embark upon a lengthy and probably boring explanation of Einstein's Theory of Relativity. However, if we were to restrict the spaceships in a space simulation to sensible speeds, the vast distances separating even the closest stars would play havoc with the mechanics of the game.

But who really cares about the accuracy of a computer game? After all, we don't need to believe that our planet is being invaded by flying saucers in order to enjoy a good game of Space Invaders. So what's the problem if we have to bend the rules occasionally? That's where

most of the fun comes in! Show our same respectable physicist the latest version of Elite, and I'm sure he'll enjoy himself no end.

Alternatively, if our game is a perfect simulation but is boring and difficult to play, then it's obviously going to be a total flop. And all our hard work will be completely wasted! When the going gets tough, we should therefore cheat shamelessly! If we can simplify the game world for our own convenience without hopelessly distorting the behaviour of the simulation, then we should do it! This especially applies to the various speeds and distances used in our calculations. In practice, few people are going to bother about the exact figures we have used in our program. So given the choice between authenticity and entertainment value, it's essential to err completely on the side of playability.

While we're designing our game, it's also important to think very carefully about precisely which aspects of the situation we are actually attempting to simulate. Normally, the game world will be vastly simpler than the world around us. Otherwise, we'd never stand a chance of dealing with it. It's therefore essential to concentrate our efforts on the interesting parts of the game, ignoring any boring or irrelevant details.

How many golfing simulations include the movements of the caddy? And although Elite has you rushing across the galaxy for *months* at a time, there's no simulation of our character eating lunch or going to the bathroom. Of course, nobody would really go to those extremes! But once we've started working on our model it's surprisingly easy to get carried away, and continue to add more and more detail, forgetting about the eventual consequences to the game.

6.2.2 The concept of game-time

Since the game world only exists inside the computer, the time which passes in the simulation is totally separate to that in the real world. For the purposes of my discussion, I'll assume game-time to be the time apparently spent in the game world, and real-time to be the actual time experienced by the player.

In games like golf, the two timescales will be pretty near identical, but if we are going to simulate the economics of an entire country over a period of years, we may need to compress years of game-time into just minutes of real-time activity. By choosing the ratio of game-time to real-time carefully, we can play out complicated scenarios which would require an enormous amount of time to complete in reality, while still keeping the gameplay fast and furious.

The time in our game doesn't even need to be continuous. Most economic simulations only update the time after the player has entered each turn. A good example of this approach can be found in Balance of Power, which takes each turn to be a single year. So the length of one of these games depends entirely on the amount of time the player is prepared to spend on each move. Rather like chess.

6.2.3 Integers and fractions

Before going any further, it's time for a few words about the way AMOS Basic handles numbers. AMOS allows us to use two separate types of numbers in our programs: Integers and Reals. Integers are whole numbers such as 1, 5, 42, or 1000. These are the values which are used for our standard variables. So in a line such as:

```
X=10
```

X and 10 are both integers. The problem with integers though, is that they are lousy for the fractional values which crop up in many simulations.

Try typing the following two examples from direct mode:

```
X=1.5
print X
ANSWER=12*3.5
print ANSWER
```

As you can see, AMOS has automatically converted our fractional 1.5 to the nearest whole number. And *ANSWER* is 36 instead of the expected 42! This effect is particularly noticeable when AMOS performs a division.

```
Print 10/4
```

produces a value of 2 rather than 2.5

If we want to use fractional values, we'll need to make use of Real or floating point variables instead. These are far more flexible, and can easily hold decimal numbers such as 1.5 or 3.141. In order to distinguish real variables from standard integers, we always place a single # sign at the end of each variable name. So X becomes X#, and ANSWER becomes ANSWER#.

Note that AMOS reads the entire floating point system directly from a special library on the start-up disk. This library will be loaded into memory automatically the first time AMOS encounters a fractional number in one of our programs. Once it's been installed, it will then be available for the duration of our current session. If you're lucky enough to have a hard drive, you may not have noticed this, but if you're using an unexpanded machine, you'll occasionally be asked to insert your AMOS Program disk into the machine. Now you know why! Let's take the previous example, and try it with the new floating point system.

```
X#=1.5
Print X#
```

gives 1.5.

```
ANSWER#=12*3.5
Print ANSWER#
```

returns 42. (Surprise Surprise!).

Finally:

```
print 10.0/4
```

outputs a value of 2.5. Correct!

The AMOS PRINT command is actually quite intelligent. Whenever it runs across a decimal point "." in one of our expressions, it automatically performs the entire calculation using real numbers. This applies even if all the rest of our values are integers. We can demonstrate this effect using the AMOS Basic TIMER function.

T=TIMER

TIMER returns amount of time which has elapsed since the Amiga has been switched on. The time period is measured in units of a 50th of a second. So if we want to convert our time into seconds, we'll need to divide it by 50 as follows:

```
print timer/50
```

This gives the number of seconds we've been using our Amiga in the current session. But:

```
print timer/50.0
```

generates a much more accurate decimal value. As a rule:

- Integers are faster than real numbers.
- Real numbers are more accurate.

In a simulation game many of our values will be fractional. So we'll need to make heavy use of these real variables.

6.2.4 Random numbers

Another common requirement in a simulation is the ability to generate unpredictable or random lists of numbers. As you may remember, AMOS Basic provides us with a special RND function for just this purpose:

r=RND(range)

RND produces a neat spread of numbers between 0 and *range*. This can be used to generate anything from the statistics in an RPG, to the weird and wonderful planet names in a space simulation. For example:

```
Print RND(5)+1
```

simulates a boring six sided dice.

```
C=RND(25)+ASC("A")
Print CHR$(C)
```

chooses a letter from A to Z.

Before using RND, we need to prime it with a seed value using the RANDOMIZE command. The best results are obtained by adding the following line to the start of our programs.

```
RANDOMIZE TIMER
```

Here a few examples of RND in action.

First we'll generate some percentage values:

```
Rem Percentages
Rem Generates a series of ten numbers from 0 to 100
For R=1 to 10
Rem print a number from 0 to 100
Print Rnd(100)
Next R
```

Now for some random letters.

Note:

=CHR$(n)
Converts a number in the range 0 to 255 into the appropriate ASCII character.

=ASC(N$)
Returns an ASCII code for the first character in *N$*. It's the exact opposite of the CHR$ function.

```
Rem letters
For L=1 To 100
Rem Get a random letter from "A" to "Z"
Rem The ASC function converts a letter into a code from 0 to 255
C=Rnd(25)+Asc("A")
rem The CHR$ function converts a number from 0 to 255 into the
Rem equivalent character.
Print Chr$(C)
Next L
```

We can also adapt the same process to generate random names. Obviously, most of the results will be garbage, but if we run the program for a while, we'll usually find a few exciting names which will be suitable for our games. These can be used directly for the planets of a space simulation, or the character names in an adventure.

This idea isn't new of course. SF writers have been employing it for years in order to generate the mysterious alien languages used in their SF novels!

```
Rem names 1
Dim NAME$(10)
For N=1 To 10
Rem Empty NAME string
NT$=""
rem generate 2 to 9 character names
For L=1 To Rnd(8)+1
  C=Rnd(25)+Asc("A")
  NT$=NT$+Chr$(C)
Next L
NAME$(N)=NT$
Print NAME$(N)
Next N
Rem Names two. This ones a bit more intelligent
Dim NAME$(10)
For N=1 To 10
Rem Empty NAME string
N$=""
Rem generate 2 to 8 character names
For L=1 To Rnd(2)+2
  Rem choose a consonant
  C=Rnd(19)+1
  Rem =MID$(X$,N,L) grabs L characters from X$ starting at N
  C$=Mid$("BCDFGHJKLMNPQRSTVWXYZ",C,1)
  Rem Pick a vowel
  V=Rnd(4)+1 : V$=Mid$("AEIOU",V,1)
  Rem Create a composite word
  N$=N$+C$+V$
Next L
NAME$(N)=N$
Print NAME$(N)
Next N
```

These demonstrate just some of the possibilities. Have a play with the routines and see what you can come up with. In a short while, we'll get on to the serious simulation stuff. But I couldn't leave this section without a short detour to one of the classic problems of simulation. Anyway, there's a game in it.

6.2.5 Simulating a deck of cards

At first glance, simulating a pack of cards is pretty easy! Each card can be assigned a single number from 1 to 52. So the Ace of Spades can be card number 1, the Two of Spades can be card 2, up to the King of Diamonds (card 52).

We can now just call up the RND function with:

```
CARD=RND(51)+1
```

This generates a random card value from 1 to 52. Unfortunately, although this idea works fine on paper, it fails miserably when we try to use in an actual game! Since each number has a precisely equal chance of being returned, there's nothing stopping this system from drawing the

same card several times in succession. Whoops! Fortunately the solution is equally straightforward. All it takes is a bit of lateral thinking. Instead of calling RND to deal our cards, we simply load the card values into an array, and shuffle them! Here's how the system works:

First, we create an array to hold our card numbers. We'll call this *CDECK*.

```
Dim CDECK(52)
```

We then fill this array with each card value in turn, starting from card 1 and ending at 52.

The CDECK array now looks like this:

CDECK(1)=1
CDECK(2)=2
: :
: :
CDECK(52)=52

Finally, we can shuffle the deck with a routine such as:

```
SHUFFLE:
For S=1 To 1000
  Rem Pick a card, any card
  S1=Rnd(51)+1
  Rem Pick another card
  S2=Rnd(51)+1
  Rem Swap the cards around
  Swap CDECK(S1),CDECK(S2)
Next S
Return
```

SHUFFLE repeatedly picks two card numbers at random and swaps them around in the deck. After a few moments, we'll be left with a randomly ordered list of numbers from 1 to 52. Perfect for our cards! We can now deal our cards out to the player. This can be handled with a routine such as:

```
DEAL:
Add TP,1,1 To 52
CARD=CDECK(TP)
WORTH=(CARD mod 13)
SUITE=CARD/4
Return
```

TP holds the position of the top card in the pack. When *TP* reaches the final card the sequence repeats from the start. This simulates the effect of returning all the cards to the bottom of the deck after each round. If you'd prefer a completely random sequence, you can easily reshuffle the pack at the end of each run.

Just replace the Add command with:

```
Inc TP: If TP>52 then Gosub SHUFFLE:TP=1
```

CARD returns the identification number of the chosen card. This number can now be used to display either the card name, or the appropriate picture from the sprite bank. For example:

```
Rem Prints the name for a predefined CARD$ array
Print CARD$(CARD)
Rem Displays a previously defined image from the sprite bank
Bob 1,100,100,CARD
```

WORTH holds the value of the card from 1 to 13. In the current system, the Ace is assumed to have a value of 1, and the King is worth 13.

SUITE returns a number ranging from 0 to 3 and represents the type of the card (Spades, Clubs, Hearts or Diamonds).

If we were displaying our cards using Blitter objects, we'd need to take special steps to reduce the amount of memory consumed to a bare minimum. Typically, we'd need to draw each card on a single 64 by 64 image. This would take around 2K per card, or 104K for the entire pack! Fortunately, we can easily compress these images using the SQUASHER utility of the AMOS Program disk. We'd now be able to hold all our card images in directly in memory, using a fraction of the original space.

Another possibility would be to draw up all our cards using the AMOS TOME map definer (available separately). TOME provides us with useful objects known as Briks. As these are built up out of a number of smaller tile elements, they use very little memory. So they're ideal for our card games.

6.3 Creating a Simulation Game

Faced with the need to generate a complete mathematical description of some real-world event, your first reaction may be to gibber uncontrollably. Relax! It's really much easier that it looks. Most models are actually created by a combination of sheer guesswork and basic trial and error. Thankfully, since we're only writing a computer game, there's no need to be particularly accurate. All that matters is that our imaginary world is believable, and that the final game is enjoyable to play. Anything else is irrelevant!

6.3.1 Defining the simulation

The first step is to decide precisely which the events we are trying to simulate. As I mentioned earlier, this can have far reaching effects on the playability of our game. Occasionally, we'll need to simulate a whole range of different activities at once. Take a space game such as Elite or Federation of Free Traders. These simulate the following events:

- ❑ The motion of one or more objects through three dimensional space (Look out for the new AMOS 3D!)
- ❑ Ship to ship combat
- ❑ Interstellar travel (possibly using hyperspace)
- ❑ Trading in a vast imaginary universe
- ❑ Interstellar politics.

In a strategy game like Supremacy, the emphasis is on the political situation. So we'd tend to concentrate on things like:

- ❑ Galactic economics
- ❑ Commerce between the various worlds in the galaxy
- ❑ Interstellar exploration
- ❑ Combat between whole fleets of ships
- ❑ Mining
- ❑ Terra-forming (a fancy SF name for making a planet habitable)

- ❑ Population growth
- ❑ Economic warfare
- ❑ The tactics of the opposing player (this can be pretty tricky!).

Each aspect of these simulations will need to be managed by its own separate set of equations. These will be combined to form the mathematical model required for our eventual game. The precise nature of the equations will obviously vary depending on the type of simulation.

Sometimes, the required model already exists and can be lifted straight out of the appropriate text book. A good example of this can be found in the 3D movement routines in games like Elite. The equations describing the motion of an object through space were first derived by Sir Isaac Newton in the 17th century. Nowadays they can be found in any decent book on A-level Physics.

Unfortunately, our local library is hardly likely to be full of books on the subject of interstellar politics. There is, however, a wealth of material on related topics such as Economics and Political history. These books may be rather dry, but they can be invaluable when it comes to designing the equations used in our simulation. They're also an excellent source of game ideas as well. Imagine a simulation of the expansion of the railroad in the Wild West. Or a game based around the long-haul sailing ships of antiquity. History is packed with examples of adventure and excitement which are literally begging to be made into a successful game. There are thousands of possibilities to choose from. Compared to the voyages of Columbus, space simulations like Elite actually seem rather tame.

In the final analysis though, no matter how much research we do on a subject, we'll still be forced to derive most of the equations in the mathematical model ourselves. This may seem a pretty tall order, but like most things, mathematical modelling gets easier with practice. Honest! Remember, since we're only writing a game we don't need to bother with the actual reality of the simulation. And it doesn't matter a jot if we initially get all the equations completely wrong!

Experience has shown that it's impossible to predict the precise behaviour of most simulations on paper. So the only means of testing our simulation, is to run it through the computer and see what happens. If the results are not acceptable, we can then go back to the original equations and make the appropriate corrections. Creating a working simulation is really just a question of trial and error. Providing we've a decent idea of the expected results, we can modify our model again and again until it is absolutely right. The snag of course, is that there's often no simple way of deciding what are the most appropriate results of a particular situation. In a political simulation, how can we ensure that the game is actually winnable in all cases?

Well, the only solution is to play the game and see what actually works out in practice. After we've played the game for some time, we can usually spot the problem points and correct them with a little inspired hackwork. Occasionally, we may need to simplify the model dramatically in order to keep things playable. As always, the more people we can con into play-testing the better. There's no real substitute for other people's perspectives, and even the best programmers have blind spots!

6.3.2 Creating a mathematical model

So much for the theory. Now for a demonstration of the creation of one of these models in practice. As an example, I'll take the trading simulation from a game like Elite. Let's assume that we wish to simulate the economics of a small vessel, trading goods from one region to another. It doesn't really matter whether the voyages take place in the oceans of Earth, or in the depths of interstellar space. The mechanics of the simulation will be identical.

The starting point is to isolate the various factors which will control the outcomes of our various events. We don't need to get these right first time. After we've worked on the model for a while, new factors will occur to us, and these can be added as required. Here are the basic factors we'll need for our trading game:

CARGO SPACE:
This is the total volume of the cargo bay. It's usually measured in units of a single ton.

FREE CARGO SPACE:
Represents the current amount of cargo space available in the hold.

CASH:
Provides the capital which will be used to fund the player's purchases. It's often valued in Credits but it could be Pounds, Dollars, or Roubles!

PURCHASE PRICE:
Holds the price at which the trader buys each unit of a particular commodity.

RESALE PRICE:
This is the amount of money which the trader can sell each ton of goods at the end of the journey. Obviously the RESALE PRICE should be greater than the PURCHASE PRICE, as otherwise the player will lose money on the deal.

BOUGHT:
Sets the amount of a particular item which will be bought at the start of the voyage.

SOLD:
This is the quantity of goods which will be sold at the conclusion of the trip.

AVAILABILITY:
Determines the number of units of the stock available at the current trading area. As a rule, the higher the availability, the lower the price.

PROFIT:
As you'd expect,this is a simple measure of the amount of money which will be made by trading sold tons of our goods at the destination point.

Armed with these factors, we can now have a first crude stab at generating the rules which will govern the simulation. In order to do this, we must first work out which factors are connected. This will enable us to make an initial guess at the required equations. Eventually, we should end up with a list like:

PROFIT:
Depends on BOUGHT, SOLD, PURCHASE PRICE, and RESALE PRICE. If a profit is going to be made on our deal, the RESALE PRICE must be greater than PURCHASE PRICE. Remember, the basic rule of trading is simply: *buy low, sell high.*

CASH:
Determined by the current value of CASH and PROFIT.

BOUGHT:
The maximum amount of goods purchased depends on CASH, PURCHASE PRICE and FREE CARGO SPACE. And the player's orders of course!

SOLD:
It's up to the player to decide how many units of a particular product should be sold at the destination. But we plainly can't sell more goods than we originally purchased. So we'll need to take account the initial value of BOUGHT.

PURCHASE PRICE:
Largely depends on the AVAILABILITY of our item at the place of purchase.

RESALE PRICE:
Similarly, this is determined by the AVAILABILITY of the goods at the selling point. Usually this value will be different from that at the place where the goods were bought.

FREE CARGO SPACE:
This will vary on the amount and volume of the goods in the cargo hold. Clearly, this value must always be less than or equal to the CARGO SPACE

We are now ready to make an initial guess at the equations which connect these factors.

PROFIT=SOLD*PURCHASE PRICE – BOUGHT*RESALE PRICE

CASH = CASH + PROFIT

BOUGHT <= CASH/PURCHASE PRICE

BOUGHT <= FREE CARGO SPACE

SOLD <= BOUGHT

Note that <= means Less than or equal. These types of rules are usually controlled with a series of explicit IF..THEN statements in our program.

FREE CARGO SPACE = FREE CARGO SPACE – BOUGHT + SOLD

This calculation will need to be repeated after every transaction. So we'll be able to keep continual track of the spare cargo space in the hold.

It's now time to calculate the PURCHASE and RESALE prices for the chosen commodity. As I mentioned earlier, these values will depend on the quantity of goods which are available at the appropriate trading area. The laws of supply and demand dictate that the price of an item will be highest when the commodity is at its rarest. Conversely, the more goods there are on the market, the lower will be the eventual price. That's really just common sense. We'd naturally expect to pay more for a glass of water in the Sahara desert than in darkest Macclesfield!

This type of connection is quite common in mathematics. It's known as an Inverse proportional relationship. The standard way of generating one, is to divide our factor into a constant.

PRICE = CONSTANT/AVAILABILITY

If CONSTANT remains fixed, the PRICE will now be inversely proportional to the value of AVAILABILITY. When the AVAILABILITY improves the price will drop, and as it falls, the PRICE will go up.

On the other hand, if we increase the CONSTANT, and keep the AVAILABILITY fixed, then PRICE will rise accordingly. This is an example of a direct proportional relationship.

The size of the CONSTANT is obviously fairly important. It's actually an indication of the intrinsic worth of the goods in question. This reflects the fact that not all items are equally valuable. Surprisingly enough, the WORTH of an object varies depending on both our location and on the particular circumstances.

If, for instance, we were dying of thirst in the Sahara desert, we'd rather find a flask of water than a £50 note, even though their rarity value might be identical. The money would be worthless, because there would be no nearby shops to spend it in. But imagine our reaction if we found the same note on our local high street! We wouldn't complain bitterly that what we really wanted was a drink of water, would we?

Sometimes a single technological development can transform the WORTH of a commodity out of all recognition. A few hundred years ago, many farms in America were contaminated with patches of a horrible tarry substance which bubbled straight out of the ground. At the time, this stuff was almost completely valueless, as the petro-chemical industry was still in its infancy. Nowadays, we call it Crude Oil and use it as the power source of our entire civilisation. Funny how times change!

Let's have another attempt at the equations of PURCHASE PRICE and RESALE PRICE.

PURCHASE PRICE = WORTH/AVAILABILITY

RESALE PRICE = WORTH/AVAILABILITY

In order to make these equations work, we'll need to ensure that the values of WORTH and AVAILABILITY, for each commodity and at each region, make sense.

Normally, we'll want all prices to be measured in whole units. This will avoid the need to make complex calculations when we are playing the game. Since we're the boss, we can use any values we like.

But even if we design things to perfection, we'll still need to experiment a little to get sensible looking results. Another point to remember is to check for the possibility that the AVAILABILITY is zero! This can be trapped with a simple IF..THEN statement in our program.

Unfortunately, there's still something missing from the current model. What would happen if we were to buy some goods and then attempt to sell them straight back to the original vendor? We wouldn't honestly expect to get the same money we paid for them would we? Any sensible retailer will demand to make a handsome profit on each transaction. Otherwise, what's the point of doing business? This can be taken into account by expanding the equations like so:

PURCHASE PRICE = PROFIT + WORTH/AVAILABILITY

RESALE PRICE = WORTH/AVAILABILITY – PROFIT

When we are trading, the values of WORTH and AVAILABILITY will vary from place to place. I'll now adjust these equations to make this a little clearer. The final equations are:

PURCHASE PRICE = PROFIT + SOURCE WORTH/SOURCE AVAILABILITY

RESALE PRICE = DESTINATION WORTH/DESTINATION AVAILABILITY – PROFIT

Obviously, if we had the time, we could expand these equations still further by adding in extra factors such as PRODUCTION COST. We could also include a random element to ensure that each region has its own separate set of prices. This could be added to either the WORTH, AVAILABILITY, or to the final prices. It would mimic the changing effect of market forces on the local economy.

The key to creating a successful trading simulation is to organise our game world so that the values of WORTH and AVAILABILITY make logical sense. In the game Elite, the planets are grouped according to the type of commodity they produce, and their economic and social sophistication. These attributes determine the WORTH and AVAILABILITY of the various commodities on the market. On an agricultural world, for instance, food is cheap and

machinery is relatively expensive. The reverse will be true at a major industrial centre. The WORTH and AVAILABILITY will vary according to the following factors:

AVAILABILITY:
Depends on the production cost for the goods at the current region and on the local demand. Agricultural regions will have a high availability for food, and industrial areas will produce large amounts of machinery.

WORTH:
This depends on the strength of the local economy, the need for the product, and the source of the goods. The strength of the economy affects the market for expensive extras such as jewels or gold. Luxuries for example, will be worthless in a poor region, and much sought after in a rich community.

The need for the goods is also variable. Supposing the current region was beset by plague. Medicine would be an extremely valuable commodity in this situation.

Often, the source of the goods will have a major impact on their perceived WORTH. Genuine Scotch whisky will always be more valuable than its Canadian equivalent, and Japanese electronics trade for a considerable premium over the identical products built in this country. The precise equations for WORTH and AVAILABILITY therefore depend on the commodity, and the number and types of the various destination points.

6.3.3 Practical considerations

Let's have a look at the way these values would be stored in the computer. The hold is easy! We just define a large two-dimensional array to store the item numbers and the amounts.

```
Dim CARGO(max_items,2)
```

But the planet data is a bit more difficult. Each area would have its own set of commodities, and these would be need to be held in an array. We might decide to generate all the values in advance, and dump them in a massive three-dimensional array such as:

```
Dim PLANET(max_planets,max_items,attributes)
```

max_planet would hold the maximum number of planets in the current game universe.

max_items would be set to the maximum number of commodities such as food, liquor, minerals and so on.

attributes would store the information which would be used to set the purchase and selling prices of our goods. It would include values like AVAILABILITY and WORTH, along with any additional factors we dreamed up.

One obvious problem with this strategy is that it costs a great deal of valuable memory. Supposing each planet had 20 commodities, and 10 attributes. This would require: 20*10*4=800 bytes per planet

So 100 planets would take 80K. What's more, we'd now have to generate all this data inside our program. Even with the help of the RND function, it would still be a pretty hefty task! A better solution would be to hold just a couple of values for each trading area. These could be used to derive all the commodity and attribute lists automatically, whenever our trading vessel entered a new region.

Here a few standard factors which could be used for this purpose:

SPECIALITY:
Represents the inherent nature of the current economy. Are the people farmers, miners, engineers, scientists, explorers or empire builders? Or are they just decadent fun lovers concerned with having a good time? This would have a tremendous effect on the goods on offer. We could easily define a separate subroutine for each type, and generate our numbers accordingly.

POLITICS:
Indicate the political stability of the region. Is the area infested by pirates, or undergoing a messy civil war? Is it a democracy, a communist society, or have the people given up and allowed computers to make all the decisions? Are certain goods illegal? If so, what's the possibility of running a successful smuggling operation? And what's the risk of getting caught?

POPULATION:
Clearly has a significant effect on demand. The more people on the planet or area, the higher the need for food and raw materials.

WEALTH:
Some societies are naturally wealthy and import large quantities of luxuries from their neighbours. Other areas don't have the resources to feed their own people. The price of expensive items such as jewellery and exotic liquor, depends almost entirely on the amount of money in the system.

TECHNOLOGY LEVEL:
How advanced is the current civilisation? Are they primitive savages, or god-like beings able to transform entire planets with a single thought?

Each of these factors would have its own separate routine in our program, rather like the character generator in an RPG. We could store them in a simple array such as:

```
Dim PLANET(max_planets,factors)
```

This would take just a fraction of the space of the previous system, and it would also add a vital element of variety into the game. When our trading vessel visited a planet, we could generate the commodity values and load them into a separate TRADING array like so:

```
Dim TRADING(max_items,attributes)
SPECIALITY=PLANET(P,0)
On SPECIALITY Gosub FARM,MINE,SCIENCE,..
POLITICS=PLANET(P,1)
On POLITICS Gosub ANARCHY,CIVIL_WAR,DEMOCRACY,...
```

Each routine would modify the previous values in the TRADING array. So we'd end up with a random list of trading values which could be used for all our various dealings.

One word of warning though. All prices should only be recalculated exactly ONCE when the trading vessel moves to a new region. Although the AVAILABILITY will change after each transaction, it should NOT be used to update the current price. This will make it too easy for the player to speculate on the market. That's because the PURCHASE PRICE will automatically drop whenever an item is sold. Similarly, the RESALE PRICE will correspondingly rise when the same goods are bought from the market.

If you think about this for a moment, you'll quickly realise that it allows the player to make an instant profit by repeatedly buying and selling a single rare commodity. I'm shocked to say that I actually found this problem in a commercial game (NOT Elite!). I was able effortlessly to equip my ship with all the latest hi-tech weaponry without ever leaving my original planet!

This type of problem is typical of the snags we can discover in even the best simulations. We might decide to solve it by increasing the values of AVAILABILITY or PROFIT so that the effect was insignificant. However, this would contribute absolutely nothing to the enjoyment of the game. We'd therefore be better off concentrating our energies on more productive territory, and leaving the AVAILABILITY fixed for the duration of our trading session.

6.4 Economic Simulations

We'll now have a look at Economic simulations which form the basis of games like Supremacy and Sim City. Although these games look very different, they all follow the same general pattern.

Economic simulations follow the progress of a complex society over a number of years. The game is divided into a series of turns, each representing a particular period in the game-world's history. At the start of each turn, our players are presented with a full breakdown of the current situation. They must now make the crucial decisions which will determine the entire future performance of the economy. An awesome responsibility! Here's a simple breakdown of the procedure:

```
Repeat
 Enter the orders from the human player
 Calculate the orders of the computer player (if it exists)
 Simulate results
 Check for victory conditions
 Test for a loss
 Next turn
Until play has won, lost, or the simulation is over
```

The interval between consecutive turns naturally varies depending on the game, but it could be anything from a month to 100 years of internal game time.

In practice, the simulation process can be divided still further. Each turn is split up into a number of separate phases. Some are performed directly by the computer, whereas others are the total responsibility of the player.

6.4.1 The phases of a simulation game

PRODUCTION:
Handled by the computer towards the end of each turn. This is the bulk of the simulation work required by our program, as it calculates the quantities and types of all the goods and resources which have been produced by the economy in the current turn.

RESOURCE ALLOCATION:
Forms the heart of the simulation, as it provides the basic mechanism by which the player controls the game. By carefully choosing which resources are to be assigned to which area, the player can manage the economy and dramatically affect the results of the simulation. Some resources such as food may be allocated immediately, whereas others such as factories will take a number of turns to come into play.

TRANSPORT:
If the game world is quite large, resources may need to be transported from the area they have been produced, to the regions which desperately need them. The player has the responsibility of setting the amounts and types of goods which are to be moved to a particular area. Once the player has given the orders, the computer then simulates this movement in the game world. Depending on the distances between the regions, each cargo may take several turns to arrive at its intended destination.

COMBAT:
This varies from game to game, but many simulations pit the player against an opposing empire of some sort. Combat occurs whenever two opposing armies meet, and it's usually resolved with the minimum of calculation. Unlike an RPG, the affects on individual soldiers are insignificant. All that really matters is the result of the battle. So the combat routine is often fairly simple.

VICTORY CONDITIONS:
When we are creating our simulations, we need to pay special attention to the scoring system. This is the method by which the players are able to gauge the success or failure of their intended strategy. There also needs to be some clear distinction between winning a game and losing it! That's the point of the victory conditions.

Most simulation games set out a number of victory conditions in advance. Their precise nature varies from game to game. Sometimes, the only measure of success is to survive the full term without getting lynched. In other games, the requirement is to completely destroy the opposing player. Occasionally though, the goal is much more clear-cut. Supremacy, for instance, gives us the specific goal of conquering the enemy's home base.

I'll now take each of these phases in turn, and discuss the various strategies which can be used to make them work in our games.

6.4.2 Resources

The nature and type of the resources will entirely depend on the world we are simulating. But they'll always have a major impact on the PRODUCTION and RESOURCE ALLOCATION phases. So here's a detailed list of the various resources which can occur in a simulation game. Don't get overwhelmed by the vast number of possibilities. Many games use only a fraction of these resources!

POPULATION:
This is a measure of the number of people which have been born or died since the last turn. It is determined by several factors including:

FOOD SUPPLY:
If the food supply drops below a certain level, people starve and the population plummets. But if food is plentiful, the population tends to increase.

LIVING CONDITIONS:
The better the living conditions, the higher the morale. Happy people have more children than depressed ones. Well, that's the assumption anyway.

TAX RATE:
If people are taxed out of existence, they will become poor. So there'll be no spare money to raise a family.

CRIME RATE:
If the inhabitants of the game world are depressed and unhappy, they may not want to bring more children into the brave new world we are creating for them!

POLITICAL SITUATION:
If the people are oppressed by their government, morale drops and the population tends to decrease.

WAR FOOTING:
During a war people die, and the population falls. After a war, there's usually a sudden, and dramatic upturn in the birth-rate. It's called a baby-boom!

FOOD:
This represents the amount of food which has been grown since the previous turn.

FARMLAND:
Indicates the net increase or decrease in the territory used for food production. The amount of farmland has a massive effect on the total food supply. If it gets too low, our people will starve!

MONEY:
Well, you know the saying! Money makes the world go round. Personally I don't believe a word of it! Gravity makes the world go round. Money just makes the experience more enjoyable. But the old adage is certainly true in a simulation game. The amount of money in circulation varies depending on the **TAX RATE** and the **POPULATION**, but it can also be increased by external trading.

If the player sells off a lot of spare resources, the money in the treasury's coffers will naturally increase. Alternatively, if the economy's doing badly, the MONEY supply could fall. If it reaches a negative value, the nation will be in debt. This isn't necessarily a catastrophe. Many nations, including Britain and the U.S.A, are now permanently in debt. In the case of America, the total national debt comes to billions of dollars! Yet America is the richest nation on Earth! Such is the curious nature of modern Economics!

MANUFACTURED GOODS:
Could be anything from a car to a computer.

LUXURY GOODS:
TV sets, videos, computer games. Anything to keep the people happy!

ARMAMENTS:
Used to expand the economy's war machine, and prepare the ground for conquest!

FACTORIES:
Used to create any of the previous three commodities. Generally they take several turns to produce, and considerable amounts of resources. So it's essential for the player to decide early on precisely what will be needed by the economy.

ARMED FORCES:
Measures the number of people who have willingly, or unwillingly, joined up since the previous year.

TRANSPORTATION:
Represents the net change in the amount of transportation available for the movement of goods from one region to another. The nature of this transportation could be anything from a railway to a space ship. Either way, it's up to the player to generate enough transportation to keep the economy firmly on its tracks.

FUEL and RAW MATERIALS:
These are used to create our products and fuel the manufacturing process.

MINES:
This is an indication of the number of new mines set up by the player. As the game progresses, the existing mines will run dry, and a continual search will be required to find the resources needed to power the economy.

6.4.3 Resource allocation

If all this sounds hopelessly confusing, then I've good news. Many of these factors are of only marginal importance. The really vital resources are:

PEOPLE:
Without people, there's no economy. It's as simple as that! Maybe I was wrong about gravity. Maybe it's PEOPLE that make the world go around! The aim of many games is therefore to manage the economy so that most of the people remain reasonably happy. This involves regulating important resources such as food, living conditions, the tax rate, the crime rate, the political situation, and the current war footing.

Even if we are playing the part of a brutal dictator, we still need people to fight our wars, or fly our battle cruisers. And if we oppress them ruthlessly, the birth rate will drop below sustainable levels, and the empire will collapse! Put bluntly, people are the most valuable resource in a simulation.

Except perhaps, for **MONEY!** It's all very fine and pleasant for us to spend all our time keeping the population contented, but unfortunately, we've also got the victory conditions to worry about! In many cases we'll be expected to rapidly expand the economy and build up our territory. This requires lots and lots of money, preferably in hard currency. Money can be raised in a variety of ways. We might increase the **TAX RATE**, although this would inevitably reduce our personal standing among the people. Or we could expand our production, and sell our goods on the open market. Or search for valuable resources such as oil. We could even push all our resources into a colonisation effort, conquering any neighbours who get in the way!

All these approaches have been tried in the real world at one time or another. Sometimes, they've worked, sometimes they've failed! It's up to the player to choose the correct strategy for the current situation.

Now for some mathematics for you to get you teeth into. Here are a few initial stabs at the equations governing POPULATION, MONEY and FUEL. They're not intended to be definitive by any means. But they should give you a valuable start. Note the # sign indicates that the factors are fractional, and will require us to use floating point variables in our calculations rather than the standard integers.

MONEY = MONEY + TAX#*POPULATION + EXTERNAL_TRADING

TAX# is a number between 0 and 1 which represents the proportion of a person's income taken by the government. The standard rate in the UK is about 0.25.

EXTERNAL_TRADING is basically the balance of payments. It's the amount of money earned or lost by the economy in a single turn.

BORN# = PEOPLE*HAPPINESS#

HAPPINESS# is a new factor I've invented on the spur of the moment. It indicates the optimism of the population and is calculated by:

HAPPINESS#=FOOD_SUPPLY#*CONDITIONS#*(1-TAX#)*CRIME#*POLITICS#*WAR#

As before, FOOD_SUPPLY#, CONDITIONS, CRIME#, POLITICS# and WAR# will be fractional values representing the effect of a particular factor on the population growth. The sizes depend on the time scale of the simulation. If we're simulating the birth rate in a single year, they'll probably be fairly small (0 to 1.5). If each term represents a 10 year period, the values would obviously need to be considerably higher.

See how I've subtracted TAX# from 1 before including it in the equation. This ensures that the higher the tax, the lower the morale. It's a useful trick!

FOOD_SUPPLY#=(FOOD_PER_PERSON#)/SUBSISTENCE

SUBSISTENCE is a measure of the amount of food needed by a single person in each period.

FOOD_PER_PERSON#=TOTAL_FOOD#/POPULATION

FUEL=FUEL+MINES*RATE +IMPORTS-EXPORTS

MINES holds the total number of mining units in the game.

RATE indicates the amount of fuel produced by each mine.

6.4.4 Transport

Transport plays an important role in a great many games. Without efficient transportation, goods can't be distributed to the places they are needed, and the whole economy will grind to a halt. This effect can be sadly seen in modern Russia, which is presently suffering from a severe distribution crisis. So food is rotting in the fields, while people go hungry! Hopefully, these difficulties will be rapidly sorted out in the near future. But as you can see, the effects of getting the transportation wrong can be disastrous!

The effect of supply on combat is even more significant. A modern war machine consumes fuel and resources at an incredible rate. Unless it's kept tanked up, the strength and efficiency of the combat units drops dangerously low. In extreme cases, entire wars have been lost because of this problem. So it pays to take proper account of it in our game.

The cost of moving goods from one place to another depends on the distance to be travelled, and the nature of the terrain separating them. In order to keep track of these costs, we'll therefore need to keep some sort of map in the computer's memory. There are several approaches we can take.

The easy option to hold our locations as a simple list of coordinate values. These would be measured in units of miles, leagues or light-years depending on the game, and could be derived straight from a simple map of our game world. This should have been previously drawn out on gridded paper during the planning phases of our game.

```
Dim AREAS(regions,2)
```

We could now work out the distance, and therefore the cost of moving directly from any two points. All we'd need to do, is compare the coordinate values and calculate the straight line distance between them. Since this involves a little maths, here's a quick explanation.

Let's assume that SX,SY are the coordinates of the starting point of our journey, and DX,DY hold the coordinates of the destination.

The first step would be to work out the horizontal and vertical distances between the start and the destination. This can be accomplished using the equations:

Horizontal distance
H#=SX-DX

Vertical distance
V#=SY-DY.

Since H and W form the sides of a triangle, we can now generate the length of the journey using Pythagoras' theorem: DISTANCE#=SQR(V#^2+H#^2)

Don't worry if this looks depressingly complicated. Just take my word for it, and use the result. It's not quite as bad as it appears.

Unfortunately, this system only works if the expense of making a journey depends solely on the distance. Sometimes however, the cost and speed of a voyage varies dramatically from region to region. If two areas are connected by a fast rail-route, or a motorway, a journey of a hundred miles could cost practically nothing. But the same distance over rough jungle terrain could take days of travel time, and would therefore be far more expensive. In this case, the only recourse is to store all the travel costs in some sort of table, using a large AMOS array.

```
Dim TRAVEL(starting point,destinations,cost)
```

We'd now need to hold the distances and costs between any two points in our game world. This table would undoubtedly consume significant amounts of memory, especially if our game world was exceptionally large. Supposing for instance, we had a set up 100 regions linked together by a complex rail network. We'd now create our table like so:

```
Dim TRAVEL(100,100,2)
```

The memory required would be:

Total number of items*memory used per item

= 100*100*2*4 (Each number takes FOUR bytes)

= 10,000*8

= 80,000 bytes or characters of information. or about 80K

Actually, the storage problem would be the least of our difficulties. After all, we'd first need to create the data and enter it into our computer! Even if we generated most of the distances in our game, it would still be a monumental task. This demonstrates the real importance keeping the game world as simple and uncomplicated as possible. If we were to concentrate on major population centres such as cities, we'd probably be able to reduce the number of regions to around 20. The difference between 20 locations and 100 is staggering.

Let's have another go at our previous memory calculation:

Memory = 20*20*2*4 bytes = 400*8 bytes =3.2K!

What's more, we'd only have to generate around 800 numbers! Easy peasy!

6.4.5 Combat

Generally, the combat routines in a simulation are fairly crude. We don't care two hoots about the results of individual battles. All we need is a simple indication of which side has won, and a tally of the resulting death-toll!

In Chapter 5, we handled our combat using a system of Melee rounds. The same idea can be readily adapted for our simulation game. Here's some pseudo-code which illustrates this process.

```
Repeat
 Work out losses to the defending side
 Work out losses to attacking side
Until the attackers or defenders are all dead, or one side retreats
```

The losses clearly depend on the relative strengths of the two sides. This is determined by several factors including:

NUMBER OF SOLDIERS:
If either of the forces outnumbers the other, the combat is likely to be extremely one-sided. Unless, of course, the opposition is equipped with the latest hi-tech weaponry!

WEAPONRY:
This represents the nature and sophistication of the armaments used by each side. It's similar to the damage factor in an RPG.

ARMOUR:
Specifies of the amount of protection afforded to each individual soldier.

EXPERIENCE:
In a war, there's often a heck of a difference between a battle-hardened warrior and a green recruit. The amount of combat experience of the participants can have a dramatic effect on the outcome of the battle.

MORALE:
Similarly, the spirit of the troops will have a major impact on their willingness to fight. If the force has just lost most of its men to enemy action, it is unlikely to put up any further resistance.

These factors can be combined to generate a single number representing the overall strength of each side. Here's a typical equation for you to look at:

STRENGTH=NO_OF_SOLDERS*MORALE*WEAPONRY*EXPERIENCE

where MORALE, WEAPONRY and EXPERIENCE are simple fractions such as:

MORALE#=1.2

WEAPONRY#=1.5

EXPERIENCE#=3

This effectively works out the strength of the force in terms of a number of *average* soldiers. It takes the assumption that one experienced man with a machine gun, is the equivalent of several spear carrying savages. While this is extremely simplistic, it does generate reasonable results.

Once we've calculated the strengths of both the attacker and the defender, we can estimate the losses due to combat with a line such as:

LOSS=RND(STRENGTH-OF_OPPOSITION)/ARMOUR#

ARMOUR# is just a number which indicates the effectiveness of the protection afforded to each soldier. The precise values would need to be found by careful experiment.

We can now subtract this LOSS value from the STRENGTH. The calculation can then be repeated for both the attackers and defenders until one of their strengths reaches zero. The side with the surviving forces wins!

Supremacy is very good at this type of combat, and includes high-quality pictures of the combat as it is occurring on the screen, along with a graphical indication of the relative strengths of the two sides. While it would be wrong to base our ideas too closely on this program, it does serve as a useful reminder of the terrific results which can be obtained with a little inspiration. Supremacy currently holds pride of place in my software library. I'm a genuine fan! So if you've yet to experience this game, you've a real treat in store!

6.4.6 Communicating with the player

By now, you should be fairly familiar with the AMOS Zone instructions. These provide everything we need to enter the user's commands directly using the mouse. With a little imagination, we can create a stunning variety of into Buttons and Selection boxes. Examples of these systems can be found on the companion disk for this book

If you're into menus, don't forget that AMOS also includes a range of advanced menuing commands. See the AMOS Manual for more details. But no matter how we enter our data, we still need an effective way of presenting important information to the user. In practice, most games keep a continual record of the economy's performance, and display the results directly in the form of a graph.

Graphs and Bar charts are not difficult in AMOS Basic, and can be generated easily using the BAR and DRAW commands. There are a couple of examples to get you started on the companion disk.

6.4.7 The opposing player

In the real world, no economy exists in isolation. Most countries are faced with ferocious opposition in everything from commerce to territory. This poses a severe challenge for the budding games designer, as it forces us to create a convincing, yet capable adversary for the player.

Currently, artificial intelligence is still in its infancy, and if we were to attempt to reproduce the complete thought processes of the human player in the computer, we'd be doomed to abject failure. Fortunately, the problem in hand is far simpler! All we require is a simulated opponent with the ability to beat the player on its own terms.

Since the simulation is being performed by a mathematical model, many of the solutions can also be derived directly from the same mathematics. In fact, once we've created the initial model, we're already well on the way to creating our opposition.

The starting point is to take a long hard look at the original factors in the simulation. These will determine the key elements which underlie the success or failure in the game. Some factors will need to be maximised, whereas other should be kept as low as conceivably possible.

In the case of our economic simulation, we'll need to take the following factors into account:

FOOD: Are the computer's subjects starving?

POPULATION: Are there enough people to run the economy?

MONEY: Are there enough resources to expand?

FUEL: What's the fuel situation like?

These factors are all connected by a series of simple rules. The POPULATION depends on the FOOD, the TAX RATE and so on. Since we've already derived these rules for our simulation model, we can use them to *guess* the results of any given strategy from the computer.

We can also measure the current performance of the computer using an evaluation function. These take the values in our simulation. and transform them into a single number which can be quickly tested in our program using a series of IF..THEN statements.

In a card game, we could measure the strength of the computer's hand using lines such as:

```
If(CSUM<PSUM or CSUM<15) and CHOLD=0
   Gosub COMPUTERS_GO
Else
   CHOLD=1 : Locate 20,17 : Print "Stick!"
End If
Rem Has the player won
If PSUM>CSUM and CHOLD=1
  Locate 0,20 : Print "You won!" : PWON=1 : Inc PSCORE
End If
```

Let's rewrite this routine using evaluation functions:

```
STATUS=PSUM-CSUM
STICK=CSUM-15
If(STATUS<0 or STICK<0) and CHOLD=0
     Gosub COMPUTERS_GO
Else
     CHOLD=1 : Locate 20,17 : Print "Stick!"
End If
Rem Has the player won
If STATUS<0 and CHOLD=1
     Locate 0,20 : Print "You won!" : PWON=1 : Inc PSCORE
End If
```

The evaluation functions are:

STATUS=PSUM-CSUM

STICK=CSUM-15

If CSUM>PSUM then STATUS would be negative. So the computer would instantly detect that it was doing badly and would take immediate steps to rectify the situation.

Similarly, STICK indicates that the computer is doing fairly well, and should stick with what it has, to avoid the risk of going bust.

In an economic simulator, the evaluation function would check the performance such as GROWTH RATE, the INCREASE in TERRITORY.and so on.

GROWTH#=TOTAL_PRODUCTION/ORIGINAL_PRODUCTION

EXPANSION#=TOTAL_TERRITORY/ORIGINAL_TERRITORY

SATISFACTION#=NUMBER_OF_PEOPLE/NUMBER_OF_STARVED

HAPPINESS#=FOOD_SUPPLY#*CONDITIONS#*(1-TAX#)*CRIME#*POLITICS#*WAR#

These functions could now be used to form the basis of a primitive computer opponent.

The key to generating our opponent, is to start off VERY simple. Limit the computer to just a couple of factors, and play a few games against it. When, and if, you beat the computer, make a careful note of the strategy you've used. Now go back to the simulation, and attempt to *hard-wire* this strategy into the program. You can then rerun the program, and try to defeat it once again. As you experiment, the computer player will slowly evolve from a primitive idiot to a sophisticated and challenging opponent. Eventually, it may be even able to beat you on a regular basis!

While you're developing the game, don't forget to save copies of the earlier versions. These will be invaluable when you're putting together the final program, as they can be used to generate a number of skill levels in the game.

6.4.8 Worked example (Kingdom)

Finally, let's have a go at creating a complete simulation game from scratch. For our example, we'll go back in time and write a new version of the classic Kingdom program. This uses a simpler version of the economic simulations we looked at earlier.

Kingdom recreates the problems faced by the ruler of a small primitive culture. The players are given limited control over the economic resources of the country, and the outcome of their decisions are calculated in terms of population growth and total income.

As in many primitive societies, the population growth depends almost entirely on the available food supply. The player has the thankless task of ensuring that a sufficient number of crops have been planted in the previous year. The aim of the game is simply to survive the term of office without getting lynched by a rampaging mob of starving peasants. In order to keep things simple, we ask the player just four questions in each turn.

Question 1: How many bushels of wheat do you wish to buy?

Question 2: How many bushels of wheat do you wish to sell?

Question 3: How many bushels of wheat do you wish to plant?

Question 4: How many bushels of wheat do you wish to give to your people?

Ok, now you've seen the ground rules, have a bash at deriving the main factors we'll need to simulate. After a while, you should come up with a list like this:

FOOD_SUPPLY

POPULATION

and of course, the MONEY!

Now we've isolated these factors, we'll need to examine them carefully to see if they can be derived from anything else. This process is quite involved, but here's what I finally came up with:

GRAIN	= The quantity of grain in the storehouse
MONEY	= The amount of money in the treasury
PRICE	= The price of grain on the market
SELL	= The number of bushels sold
BUY	= The number of bushels bought

I'm assuming that there's a friendly merchant out there, who'd be happy to sell the players all the food they need, at a price!

CROP	= The number of bushels harvested this year
PLANTED	= The amount of grain planted this year.
YIELD	= Number of bushels harvested on each bushel we plant.

Note: Unless the yield is greater than about 2, all our people will starve! I discovered this fact by accident when I was working on this game for the STOS version of this book. I'd forgotten to take into account the reason people plant grain. To get more! But after a few games where all my people died, I quickly realised my mistake and expanded the model. This demonstrates an important point. Don't expect to get things right first time. It's impossible!

LOST	= The quantity of grain lost because of pests
TOTAL FOOD	= The total amount of food allocated to our subjects
FOOD	= The amount of food available for each subject
SUBSISTENCE	= The minimum amount of food required per person.

I derived this from a crazy system based on cornflakes! But the figure worked fine in the simulation!

PEOPLE	= The total number of people
GROWTH	= The number of people born/starved in the last year.

If you look at these factors very carefully, you should be able to deduce a number of general rules which will govern the simulation.

MONEY = MONEY + SELL*PRICE – BUY*PRICE

This is just basic economics.

CROP = PLANTED*YIELD

The value of YIELD is vital, as it determines the amount of food which can be raised by each ton of crops.

FOOD = (TOTAL FOOD)/PEOPLE

FOOD is the amount of food allocated to each individual person.

NEWPEOPLE = (TOTAL FOOD)/SUBSISTENCE

NEWPEOPLE holds the new population of the kingdom. In reality, people have children for a variety of reasons. In the simulation, the population expands to the limit of the available food supply. Then, when the food supply drops, they starve!

GROWTH = (NEW PEOPLE)-PEOPLE

This indicates who has been born or has died since the previous term.

SATISFACTION# =FOOD/SUBSISTENCE

SATISFACTION# provides a crude measure of the population's prosperity! I'm assuming that the most important factor in the people's minds is a full belly. Well, they do say that the way to a man's heart is through his stomach! Note that this factor also takes the growth rate into account as well. Since all the food is distributed evenly, any shortage will automatically result in a drop in the population.

These equations come directly from the factors I originally isolated, using a combination of guesswork and experimentation. Although they omit literally dozens of features found in the real world, they still provide a useful basis for our version of Kingdom.

So without any more ado, here's the program!

Example 6.1: Kingdom

```
Set Buffer 40
Rem Example 4.6
Rem The simulation model here may be crude
Rem But you can have great fun playing around with it
Rem Kingdom!
Hide On
Dim D(11,2)
Screen Open 0,320,255,4,Lowres
Paper 0 : Curs Off : Cls 0
Screen Display 0,128,70,,
Screen Open 1,330,100,4,Lowres
Paper 0 : Curs Off : Cls 0
Screen Display 1,128,70,,
Screen Open 2,320,100,4,Lowres
Paper 0 : Curs Off : Cls 0
Screen Display 2,128,170,,
Screen Open 3,320,16,4,Lowres
Curs Off : Paper 0 : Cls : Centre "<Kingdom> "
Screen 2
Gosub NEW_GAME
Gosub CHECKIT
Do
   Gosub BUYIT
   If BUY=0 Then Gosub SELLIT
```

```
   Gosub PLANT
   Gosub FEED
   Gosub TURN
Loop
Rem New game
NEW_GAME:
Curs On : Locate 0,0 : Input "What's your name?";NAME$ :
Print "Welcome to Mundania King ";NAME$ : Wait 40 : Cls : Curs Off
PEOPLE=1000+Rnd(200) : MONEY=PEOPLE*24
SPEOPLE=PEOPLE
GRAIN#=Rnd(1000)+3000 : NGRAIN#=GRAIN# : CROP=0 : SATISFACTION#=1
PLANTED=0 : LOST=0 : FOOD#=0 : DEATH_TOLL=0
YEAR=1 : GAMEOVER=0
SUBSISTANCE=3 : BUY_PRICE=8+Rnd(3) : SELL_PRICE=BUY_PRICE-2
Rem assuming 1 bushel=100kg of grain,
Rem and  2kg of grain = 1kg of CORNFLAKES
Rem Thats around one large packet of cornflakes a day
Rem What  an  unpleasant  prospect.  You  probably wouldn't starve
Rem Ok, so the derivation may be a LITTLE unorthodox
Rem But who worries about acurracy when they're having fun
LOST=0 : NEW_PEOPLE#=0 : FOOD_RATION#=0
D(1,1)=PEOPLE : D(1,2)=GRAIN#
Return
BYE: Pop : Stop
Rem next turn
TURN: Rem Set yield per bushel. Tweek as desired
YIELD#=2+(Rnd(3)/10.0)
Rem Reap Crop. I've assumed wastage is always at least 10 percent
CROP=PLANTED*YIELD# : LOST=Rnd(CROP/10+1)+CROP/10
Rem In order keep the game challenging I've assumed that
Rem wastage increases for larger crops
Rem This can be justified if large crops attract swarms of locusts
Rem That's my excuse anyway
If CROP>10000 Then CROP=CROP-Rnd(5000)
ROTTED=Rnd(NGRAIN#/3+1)
GRAIN#=NGRAIN#+CROP-LOST
FOOD_RATION#=FOOD#/PEOPLE : Rem calculate ration for each person
Rem Assume no of people rises along  with increasing prosperity
NEW_PEOPLE#=FOOD#/SUBSISTANCE : Rem calculate people at end of year
RAN#=(100+Rnd(4)-2)/100.0
NEW_PEOPLE#=NEW_PEOPLE#*RAN#
GROWTH#=NEW_PEOPLE#-PEOPLE
PEOPLE=NEW_PEOPLE#
BUY_PRICE=7+Rnd(4) : SELL_PRICE=BUY_PRICE-2
If GROWTH#<0 Then STARVED=Abs(GROWTH#) : INFLUX=0 Else STARVED=0
INFLUX=GROWTH#
STATUS:
Screen 0
Curs Off : Cls 0
Locate 0,1 : Print " In  the  year ";YEAR;" of king ";NAME$
SATISFACTION#=FOOD_RATION#/SUBSISTANCE
If INFLUX>0
   Locate 0,4 : Print INFLUX; : ARRIVED=ARRIVED+INFLUX
Else
   Locate 0,4 : Print " No    ";
End If
Locate 7,4 : Print " People  entered the city "
Locate 0,5 : Print STARVED; : Locate 7,5 : Print " People  starved"
Locate 0,7 : Print CROP; : Locate 7,7 : Print " Bushels were grown"
Locate 0,8 : Print LOST; : Locate 7,8 : Print " Bushels were eaten by
```

```
locusts"
Locate 0,9 : Print ROTTED; : Locate 7,9 : Print " Bushels rotted in the
warehouse"
Locate 0,11 : Print " Thy treasury   hath    ";MONEY;
Locate 30,11 : Print "Gold"
Locate 0,12 : Print " Thy warehouse  holds   ";GRAIN#;
Locate 30,12 : Print "Bushels"
Locate 0,13 : Print " Thy subjects   number ";PEOPLE;
Locate 30,13 : Print "Citizens"
Screen To Front 0
Wait 40
Inc YEAR : D(YEAR,1)=PEOPLE : D(YEAR,2)=GRAIN#
NGRAIN#=GRAIN# : PLANTED=0 : FOOD#=0
Locate 0,17 : Clear Key
DEATH_TOLL=DEATH_TOLL+STARVED
If YEAR<=10
   If SATISFACTION#>=1
      Print " Prosperity  reigns in Mundania" :
      Print " All hail the good king ";NAME$
   End If
   If SATISFACTION#<1 and SATISFACTION#>0.9
      Print " Thy people are hungry!"
   End If
   If SATISFACTION#<0.9 and SATISFACTION#>0.5
      Print " Thy people are starving!"
   End If
   If SATISFACTION#<0.5
      Print
      Print " Thy  people  are in revolt!"
      Print " Thy reign of terror ends at year ";YEAR
      Print " King ";NAME$;" the bad is DEAD!"
      GAMEOVER=1
   End If
Else
   Print
   If PEOPLE>SPEOPLE and DEATH_TOLL<500
      Print " After ten happy years"
      Print " Your term of office has expired."
   End If
   If DEATH_TOLL<400
      Print " Not bad! You only lost";DEATH_TOLL;" People!"
   End If
   If DEATH_TOLL>400 and DEATH_TOLL<1000
      Print " Your  term of office has Expired "
      Print " Unfortunately, so have most the people!"
   End If
   If DEATH_TOLL>1000
      Print " Your term of office finally terminated"
      Print " Survivors are dancing in the streets!"
   End If
   Print " During your rule ";DEATH_TOLL;" people starved"
   Print " And ";ARRIVED;" new people settled in the city "
   GAMEOVER=1
End If
Locate 0,25 : Centre "Press any key to continue" : Wait Key : Cline
If GAMEOVER Then Gosub GAME_OVER
Gosub CHECKIT
Screen To Back 0
Screen 2
Return
```

```
Rem buy grain
BUYIT:
PRICE=BUY_PRICE
If PRICE<2 Then PRICE=2
Cls : Locate 0,0 : Centre "Buy Grain"
OK=0
Repeat
   Locate 0,2 : Print "Grain costs ";PRICE;
   Locate 18,2 : Print " per bushel"
   Locate 0,3 : Print "You have    ";MONEY;
   Locate 18,3 : Print " pieces of gold"
   Locate 0,6 : Cline : Input "How many bushels wilt thou buy?";BUY
   If BUY=0
      OK=1
   End If
   If BUY*PRICE>MONEY
      Print "You don't have enough money"
      Wait 40 : Locate 0,7 : Cline
   Else
      MONEY=MONEY-BUY*PRICE : GRAIN#=NGRAIN#+BUY : NGRAIN#=GRAIN#
      OK=1
   End If
Until OK
Cls
Gosub CHECKIT
Return
Rem sell grain
SELLIT: PRICE=SELL_PRICE
Cls : Locate 0,0 : Centre "Sell Grain"
Locate 0,2 : Print "Grain costs ";PRICE;
Locate 18,2 : Print " per bushel"
Locate 0,3 : Print "You have    ";NGRAIN#;
Locate 18,3 : Print " bushels in the store"
OK=0
Repeat
   Cline
   Locate 0,6 : Input "How many bushels wilt thou sell?";SELL
   If SELL=0 Then OK=1
   If SELL>NGRAIN#
      Locate 0,7 : Print "You don't have enough grain"
      Wait 40 : Locate 0,7 : Cline
   End If
   MONEY=MONEY+SELL*PRICE : GRAIN#=NGRAIN#-SELL
   NGRAIN#=GRAIN# : OK=1
Until OK
Cls
Gosub CHECKIT
Return
Rem check status
CHECKIT:
Screen 1
Cls
Locate 0,0 : Print " Year ";YEAR;" of King ";NAME$
Locate 0,2 : Print " Thy treasury containeth ";MONEY;
Locate 32,2 : Print "Gold"
Locate 0,3 : Print " Thy warehouse holds     ";NGRAIN#
Locate 32,3 : Print "Bushels"
Locate 0,4 : Print " Thy subjects  number    ";PEOPLE;
Locate 32,4 : Print "People"
Locate 0,6 : Print " Thou hast planted       ";PLANTED;
```

```
Locate 32,6 : Print "Bushels"
Locate 0,7 : Print " Thy people have          ";FOOD#;
Locate 32,7 : Print "Bushels"
Screen 2
Return
Rem Plant seed
PLANT:
Cls : Locate 0,0 : Centre "Plant crops"
Locate 0,3 : Print "Thou hast ";NGRAIN#;" bushels in the store"
OK=0
Repeat
   Locate 0,5 : Cline : Input "How many bushels wilt thou plant?";PLANTED
   If PLANTED=0 Then Input "Are  you  sure?";AN$
   If(AN$="y") or(AN$="Y")
      OK=1
   End If
   If PLANTED>NGRAIN#
      Print "You  don't  have enough grain" : Wait 40 : Cup : Cline
      PLANTED=0
   Else
      NGRAIN#=NGRAIN#-PLANTED : OK=1
   End If
Until OK
Cls
Gosub CHECKIT
Return
Rem Feed people
FEED:
Cls : Locate 0,0 : Centre "Ration food"
Locate 0,3 : Print "Thou hast ";NGRAIN#;" bushels in the store"
OK=0
Repeat
   Locate 0,5 : Cline : Input "What wilt thou feed to thy people?";FOOD#
   If FOOD#=0 Then Input "Are you  sure?";AN$
   If(AN$="y") or(AN$="Y")
      OK=1
   End If
   If FOOD#>NGRAIN#
      Print "There  isn't enough grain" : FOOD#=0 : Wait 40 : Cup : Cline
   Else
      NGRAIN#=NGRAIN#-FOOD# : OK=1
   End If
Until OK
Cls
Gosub CHECKIT
Return
Rem another game?
GAME_OVER:
Screen Open 3,640,100,4,Hires : Curs Off : Cls 0
NO_POINTS=11 : MX_POINTS=11 : X=40 : Y=20 : W=25 : H=50
MX_HEIGHT#=1200 : S=1 : MT$="Population" : BT$="Years " : ST$="People"
P=3 : HSC=1 : VSC=0 : Ink 2,0,2
T=1 : T$=MT$ : Gosub GRAPH4 : Ink 2,0,2 : Gosub SCALE2
MT$="Food supply" : BT$="Years" : ST$="Bushels"
MX_HEIGHT#=8000
T=2 : X=360 : T$=MT$ : Gosub GRAPH4 : Ink 2,0,2 : Gosub SCALE2
Screen 0 : Locate 0,25 : Print "Another game" : Input A$
If(A$<>"Y") and(A$<>"y")
   Stop
Else
```

```
   Screen Close 3
   Cls : Screen 1 : Cls : Screen 2
   Cls : Screen To Back 0 : Gosub NEW_GAME : Return
End If
SCALE2:
Rem Inputs X,Y,W,H,MX_POINTS,MX_HEIGHT,SX,HSC,VSC,ST$,BT$
XA=X : YA=Y : V=MX_HEIGHT# : Gosub CNUM : YA=Y+H : V=VSC : Gosub CNUM
YA=Y+H+20+Text Base : T$=BT$ : Gosub GCENTRE : T$=ST$ : Gosub VCENTRE
For A=0 To MX_POINTS-1
   A$=Str$(A+HSC)-" " : Text X+A*W,Y+H+10,A$
Next A
Return
CNUM:
V$=Str$(V)-" " : LM=Text Length(V$) : Text XA-LM,YA+Text Base,V$ : Return
GCENTRE:
CX=W*MX_POINTS-Text Length(T$) : LX=XA+CX/2 : LY=YA-Text Base
Text LX,LY,T$ : Return
VCENTRE:
Rem inputs T$,X,Y,W,H,MX_POINTS
CH=8 : VH=H-Len(T$)*CH : SY=Y+Text Base+VH/2
For C=1 To Len(T$) : C$=Mid$(T$,C,1) : Text SX,SY+(C-1)*CH,C$ : Next C
Return
GRAPH4:
Rem Inputs D(),NO_POINTS,MX_POINTS,T$,X,Y,W,H,MX_HEIGHT#,P,S
SX=X+W*(MX_POINTS-1) : Box X,Y To SX+1,Y+H+3
XA=X : YA=Y : SX=SX+16 : Gosub GCENTRE : SF#=H/(MX_HEIGHT#-1)
PY=D(S,T)*SF# : Plot X,Y+H-PY : Draw X-2,Y+H-PY To X+2,Y+H-PY
If NO_POINTS=S Then Return
For B=S+1 To NO_POINTS
   PY=D(B,T)*SF# : Draw To X+(B-S)*W+1,Y+H-PY
Next B
Set Paint 0 : Return
```

Expansion possibilities:

- ❑ Introduce the concept of taxation to the economy.
- ❑ Expand the number and types of commodities to be managed by the player.
- ❑ Establish a shipping line, bringing new goods at regular intervals. The likelihood of supplies reaching our shores depends on a number of factors, including the season and the weather.
- ❑ Add a competitor, fighting our player for scarce resources.

If you persevere, you should be able to expand this game almost indefinitely! That's how the simulation game started, after all! Drop me a line care of Sigma if you come up with something special!

6.5 Conclusions

Simulations are one of the most challenging areas of computer gaming, but they are also one of the most engrossing. The range and depth of the subject is immense, and it's impossible to encompass the entire topic in a single short chapter. But don't be put off by its apparent complexity. If these games are within the reach of computer students using crude mainframe versions of Basic, they should be child's-play to any experienced AMOS Basic programmer. The fact is, simulation really can be fun. So go out and simulate something today!

7

Scrolling Techniques

7.1 Introduction

The ability to set the action against a rapidly moving background has now become a familiar part of everything from an arcade game to an RPG. Oddly enough, scrolling is one of those activities which has become progressively more difficult as computers have advanced. Although the Amiga is capable of displaying some stunning images, these require massive amounts of space, especially when compared to earlier machines such as the Commodore 64.

In order to scroll one of the Amiga's screens by a single line, it's necessary to copy an incredible 30,000 pieces of information around in memory. Typically, these scrolling operations would need to be performed dozens of times in a single second. This is a severe challenge for even the fastest computer.

Except of coarse, the Commodore Amiga! That's because the Amiga comes complete with a built-in Blitter chip, which allows us to move our screen data at incredible speeds. At its fastest, it can handle up to 16 million screen points per second! Whew! The Blitter really is a remarkable piece of hardware, and it's used to great effect by the AMOS Basic BOB commands to create the fast moving images required in our arcade games. But the BOB commands only make use of a fraction of the Blitter's full potential. We can also exploit it to generate a terrific range of screen scrolling effects in our games.

7.2 Blitter Scrolling (Software Scrolling)

As you would expect, AMOS Basic supports several dedicated commands, especially for our screen scrolling operations. Surprisingly however, most experienced programmers tend to ignore them completely, and use the general purpose SCREEN COPY instruction instead! Actually, the SCREEN COPY command has a number of real advantages over the competition. SCREEN COPY gives us total control over the entire scrolling process, and it's considerably more flexible than the equivalent SCROLL or SCREEN OFFSET commands. It's also murderously fast! So there's no percentage in calling the built-in screen scrolling routines. Often these just add an extra level of complication to an otherwise simple procedure.

7.2.1 The SCREEN COPY command

The SCREEN COPY command provides us with an easy way of copying large sections of an AMOS Basic screen around in memory. The format is:

SCREEN COPY source,tx,ty,bx,by To destination,dx,dy

where:

source is the number of an existing AMOS Basic screen. Remember, the standard AMOS display is automatically assigned to screen number zero. Additional screens can be created at any time using the standard AMOS SCREEN OPEN command. *tx,ty,bx,by* are the screen coordinates of an imaginary rectangular box surrounding the area which is to be copied. *destination* holds the number of an AMOS screen which will be loaded with the new image. The destination doesn't have to be a different screen incidentally. So there's nothing stopping you from copying your images straight onto the original source screen if required. Finally, *dx,dy* set the starting position on the destination screen where your data will be inserted. Note that if you are copying directly onto a single screen, the source and destination regions may safely overlap.

Ex STOS users will be delighted to find that there are absolutely NO limitations on the size and position of the source and destination areas. The equivalent STOS command only worked on 16 pixel boundaries, making it almost impossible to generate smooth horizontal scrolling effects on the ST. Yuck!

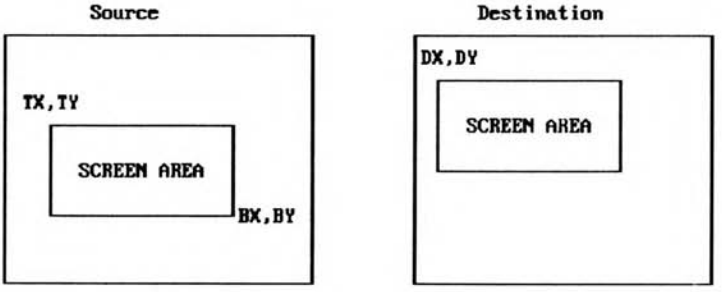

Figure 7.1: The Screen Copy command

There's also a second form of the SCREEN COPY command as well. This allows us to transfer entire screens in a single smooth operation:

```
SCREEN COPY source TO destination
```

Copies the entire contents of screen number *source* to the selected destination screen. It's equivalent to:

```
SCREEN COPY source,0,0,Screen Width,Screen Height To destination,0,0
```

Now for a small example to get you started.

```
Rem Draws a filled bar on the screen
Set Paint 1
Set Pattern 8
Bar 80,80 To 240,120
Paper 0
Locate 15,12 : Print "Amos Basic"
Rem make a copy at coordinates 80,10
Screen Copy 0,80,80,240,120 To 0,80,10
Locate 0,0 : Centre "Copy 1"
rem Make a second copy at coordinates 80,150
Screen Copy 0,80,80,240,120 To 0,80,150
```

```
Locate 0,18 : Centre "Copy 2"
```

Try changing the sizes of the source and destination rectangles of these SCREEN COPY commands. You should be able to generate a range of intriguing effects.

7.2.2 Creating a scrolling message

I'll now show you how the SCREEN COPY instruction can be used to create a simple scrolling message. During the coarse of this chapter, I'll gradually extend these routines to handle the scrolling playfields you'd need in an actual game.

The easiest technique uses separate screens to hold the source and destination areas. The source is an extra wide screen which is capable of holding a complete picture of our message in memory. It's generally hidden away in the background so as not to interfere with the main program display. Our TV picture is taken directly from the destination screen. This serves as a *viewport* on the message, and displays a few short words of the text at a time. If we progressively copy the appropriate parts of the source screen to the destination, we can steadily move the viewport through the entire message. Supposing we wanted to scroll the text as shown in Figure 7.2:

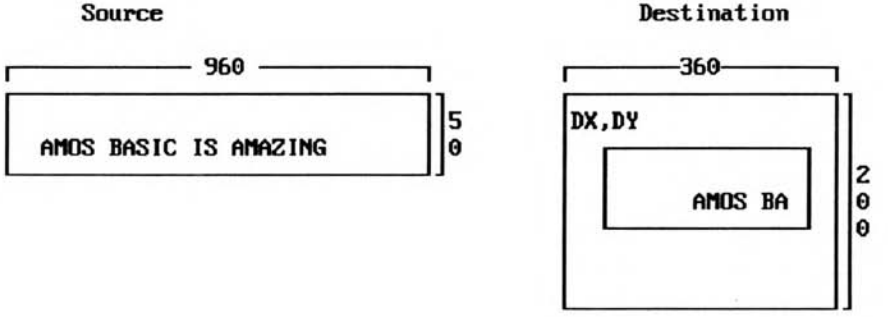

Figure 7.2: Scrolling through a viewport

We could now move through our message like this:

```
For SX=0 To 640
      Screen Copy SOURCE,SX,0,SX+320,50 To DESTINATION,DX,DY
Next SX
```

SX determines the starting point of the area to be copied from the source screen, and *DX,DY* sets the position of the viewing window on the destination. As *SX* increases, the source area moves slowly to the right. So each successive SCREEN COPY command scrolls a new section of the source screen into the viewport. Here's an example for you to type in:

Example 7.1: Scrolling through a large horizontal map

```
Close Editor
Close Workbench
Rem Create destination screen
Screen Open 0,320,200,8,Lowres
Rem I'll be explaining this in a few moments
Rem Double Buffer
Curs Off : Cls 0 : Paper 0 : Hide
Rem create source screen
```

```
Rem It's EXTREMELY wide, but not only fifty units high
Screen Open 1,960,50,8,Lowres
Rem Clear the source screen and turn off the text cursor
Curs Off : Cls 0 : Paper 0
Rem move destination screen in front of source
Rem so Screen 1 is hidden from view
Screen To Front 0
Rem create a banner on the invisible source screen
BANNER["  AMOS BASIC  AMOS BASIC "]
Screen 0
Locate 0,0 : Pen 3 : Centre "Horizontal Screen Scrolling"
Rem set up an attractive rainbow effect on the destination
Colour 2,0
Set Rainbow 1,2,32,"(2,1,16)","(2,1,16)","(2,1,16)"
Rainbow 1,0,150,32
Rem scroll the screen
Do
   For SPEED=1 To 16
      For CS=0 To 12*40-1 Step SPEED
         Rem Copy the source from CS to CS+320 onto the destination
         Screen Copy 1,CS,0,CS+320,50 To 0,0,100
         Rem These commands  are lying in wait for the next section
         Rem Screen Swap:Wait Vbl
     Next CS
   Next SPEED
Loop
Procedure BANNER[A$]
   Rem creates a CRUDE banner effect
   Screen 0:Rem use screen zero for all text and graphics operations
   Print A$
Rem Create a large image of the message text in screen 1
   Zoom 0,0,0,(Len(A$)-1)*8,8 To 1,0,0,959,32
   Rem select screen 1 again.
   Screen 1
End Proc
```

The results of this program are slightly disappointing, as there's a noticeable flicker during each scrolling operation. The problem arises because we can see the image as it's actually being created on the screen.

AMOS Basic allows us to create our images on an invisible *logical* screen before we display them on our TV or monitor. This enables us to conceal our scrolling operations from view until they have been fully drawn. Hopefully, this idea should ring a bell, as it's the same double buffering system we used for our Blitter objects.

Just to make life easier for you, I've already included the appropriate instructions into Example 7.1. Check through the listing until you find the following lines:

```
Rem Double Buffer            (Near the start)
Rem Screen Swap:Wait Vbl     (Just after the Screen Copy)
```

Now remove the REM statements to activate the double buffering system, and run the program. Hey presto! No flicker!

Before we go any further, it's worth examining this *Double Buffering* system in a little more detail.

7.2.3 Double Buffering

The DOUBLE BUFFER command creates two separate versions of the currently selected screen. The *physical* screen contains the picture which is actually being displayed on our TV or monitor. Any changes to this picture will be immediately reflected on the screen. The *logical* screen is an invisible buffer zone used for our drawing operations.

AMOS Basic provides us with two simple functions which allow us to access these screens directly from the SCREEN COPY command.

=PHYSIC(Screen number)

=PHYSIC

PHYSIC selects the PHYSICAL part of the selected screen. The screen number is optional. If it's omitted, AMOS will assume we are referring to the screen.

=LOGIC(Screen number)

=LOGIC

LOGIC uses the concealed *logical* screen instead. The copying operation is now totally invisible, and doesn't interfere with the existing program display at all.

We can enter these functions as the source or destination screens in our SCREEN COPY commands:

```
Screen Copy Physic(1) to Logic(1)
```

which copies the entire contents of *physical* screen number 1 to the appropriate *logical* screen. Alternatively:

```
Screen Copy Logic(1),CS,0,CS+320,50 To Logic(0),0,100
```

transfers a 320 by 50 section of screen 1 to screen 0, starting at coordinates 0,100. Once the double buffering system has been set, all operations will be performed on the logical screen by default. For example:

```
Screen Copy 0 to 1
```

is equivalent to:

```
Screen Copy Logic(0) to Logic(1)
```

On the other hand, if double buffering isn't used, then the physical and logical screens will be exactly the same. So the LOGIC and PHYSIC commands will have no apparent effect.

After the SCREEN COPY command has done its work, the two screens can be smoothly switched around with the AMOS Basic SCREEN SWAP instruction. The original logical screen is now displayed on our TV or monitor, and replaces the existing physical screen. Similarly, the old physical screen is hidden away in memory, and forms the basis of a brand new logical screen.

It's important to recognise that SCREEN SWAP doesn't actually MOVE anything. It simply changes the current screen display using the internal equivalent of the SCREEN HIDE or SCREEN SHOW instructions.

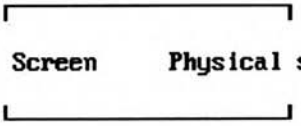

Figure 7.3: Screen swap

Note: The two screens will only be swapped during the next vertical blanking period. So we'll need to place a WAIT VBL command after the SCREEN SWAP instruction before continuing with our program. Otherwise, AMOS may have executed several further commands by the time the screens are finally swapped.

If we're double buffering several screens, we can also include an optional screen number with the instruction like so:

Screen Swap N

This swaps the logical and physical components of screen number *N*.

Normally, AMOS Basic attempts to synchronise all drawing operations precisely with the movements of our Blitter objects. This system is great for static displays, but it interferes badly with our scrolling routines. We therefore need to deactivate the system completely during our initialisation stage. We can accomplish this with a command called AUTOBACK.

AUTOBACK toggles the display between three possible drawing modes. The one we are currently interested in is AUTOBACK 0. This forces AMOS to draw all our graphics on just the logical part of the screen. When we've finished our picture, we can then load it directly into the physical screen with:

```
        Screen Swap:Wait Vbl
        Screen Copy Physic To Logic
```

Both screens will now contain identical images.

The final requirement is to remove all our Blitter objects from the screen. This stops the SCREEN COPY command from clashing with the BOB movements and causing a horrible mess on the display. The general procedure can be summarised as follows:

```
Turn off autoback system (AUTOBACK 0)
Draw our graphics in the logical screen
Load a copy of our logical screen into the physical one
(Screen Swap:Wait Vbl: Screen Copy Physic To Logic)
Kill automatic Bob updates (BOB UPDATE OFF)
Main program loop
Do
        Remove Bobs from screen with BOB CLEAR
        Perform scrolling operation
        Move Bobs
        Replace Bobs at their new positions with BOB DRAW
        Switch logical and physical screens (SCREEN SWAP:WAIT VBL)
Loop
```

7.2.4 Rolling a screen

You may have noticed that I had to cheat a little in Example 7.1. In order to get the text to scroll continuously, I was forced to load two identical copies of my message into the original source string. Obviously it would be much neater if we could rotate the entire message

smoothly across the screen. The effect we are aiming at can be seen from the diagram in Figure 7.4.

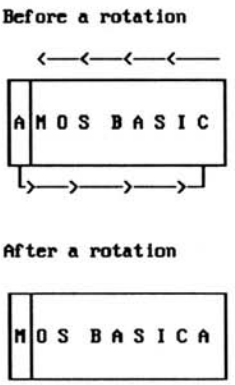

Figure 7.4: Rotating the screen

On the face of it, this rotation effect looks pretty easy. Let's assume each letter was 32 units long. We could now move the **MOS BASIC** bit into place with a command such as:

```
Screen Copy 0,32,0,320,200 to 0,0,0
```

The process and result is shown in Figure 7.5.

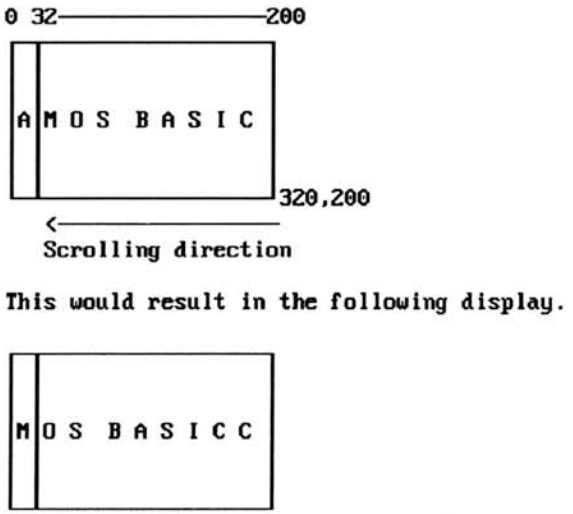

Figure 7.5: Using SCREEN COPY to rotate a screen

Whoops! Our copying operation has now overwritten the **A** in **AMOS**, and there's a extra **C** at the right of the screen. The problem arises because of the way the SCREEN COPY command works. As its name suggests, SCREEN COPY only copies our screen data from one place to another. It has absolutely NO effect on the source image. So when we attempted our rotation, the original copy of the **C** remained firmly stuck at its previous position. Clearly, we're doing something wrong. If we want to generate the required rotation effect, we'll first have to store the part of the screen containing the **A** in a safe area of memory. We can then perform our SCREEN COPY command as before, and then simply copy the **A** over the final **C**.

One possibility would be to grab the area from 0,0 to 32,200 into memory using the GET ICON or GET BLOCK commands. While there's nothing really wrong with this approach, there's actually a much neater solution. Supposing we were to expand the screen area to include an invisible *buffer zone*. Providing this zone was positioned just off the edge of the screen as in Figure 7.6, it would have no apparent effect on our display. We could then copy the section of the screen holding the **A** into the buffer area as shown in Figure 7.7.

The left hand side of the screen now holds a complete copy of our rotated image. So we can move this straight into place with the SCREEN COPY command:

```
Screen Copy 1,32,0,320+32,200 to 1,0,0
```

This would generate the image shown in Figure 7.8.

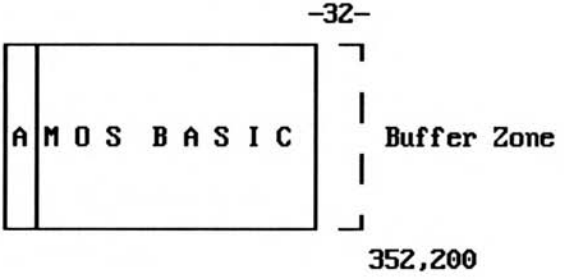

Figure 7.6: Using a hidden buffer zone

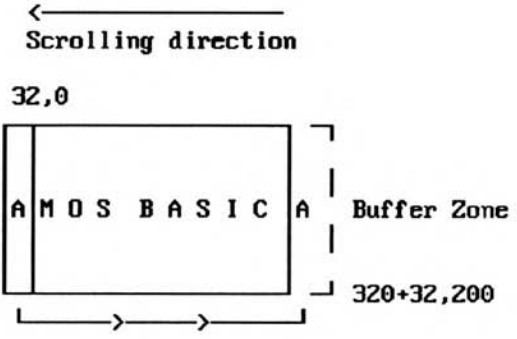

Figure 7.7: Copying the A into the buffer zone

Figure 7.8: Rotating into a buffer zone

Perfect! Although the buffer zone still contains a spurious **A** character, this doesn't matter, since it's completely invisible. If we repeat this procedure, our message will be gently rotated across the screen. This can be seen from the following example:

Example 7.2: Rolling the screen

```
Close Editor
Close Workbench
Rem Create destination screen
Screen Open 0,320,200,16,Lowres
Double Buffer
Curs Off : Cls 0 : Paper 0 : Hide
Rem create source screen 640+40 units long
Rem This allows 40 units for the buffer zone
Screen Open 1,680,50,4,Lowres
Curs Off : Cls 0 : Paper 0
Rem move destination screen in front of source
Screen To Front 0
Rem create a banner on the source screen
BANNER[" AMAZING AMOS BASIC "]
Screen 0
Locate 0,0 : Pen 3 : Centre "Horizontal Screen Scrolling"
Rem set up an attractive rainbow effect on the destination
Colour 2,0
Set Rainbow 1,2,32,"(2,1,16)","(2,1,16)","(2,1,16)"
Rainbow 1,0,150,32
Rem scroll the screen
Do
   For SPEED=1 To 16
      For CS=0 To 800
         Rem rotate the source screen
         Screen Copy 1,0,0,SPEED,50 To 1,640,0
         Screen Copy 1,SPEED,0,SPEED+640,50 To 1,0,0
         Rem Copy the source screen to the destination
         Screen Copy 1,0,0,320,50 To 0,0,100
         Screen Swap
        Wait Vbl
      Next CS
   Next SPEED
Loop
Procedure BANNER[A$]
   Screen 0 : Print A$
   Zoom 0,0,0,(Len(A$)-1)*8,8 To 1,0,0,639,32 : Screen 1
End Proc
```

You may be wondering why I'm rotating the message on a separate screen, and then laboriously copying the rotated image onto the main display. Isn't that rather wasteful? Well, it all boils down to a problem with the logical and physical screens we encountered a few pages

ago. In order for our scrolling effects to be really smooth, each operation has to follow directly from the previous one.

Let's see what would happen if we ignored this problem, and performed all our scrolling operations on the original source screen. At the start of our program, we'd first load the logical and the physical screens with identical pictures of our message.

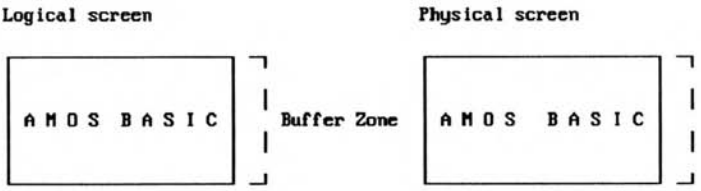

Figure 7.9a: Loading the logical and physical screens

We'd then rotate the logical screen using our standard scrolling routine, and flick this neatly onto the display with the SCREEN SWAP command.

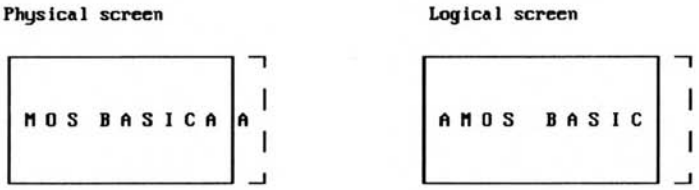

Figure 7.9b: Rotating the logical screen

The logical screen would now contain an exact copy of the original message. This would be rotated to get the result in Figure 7.9c.

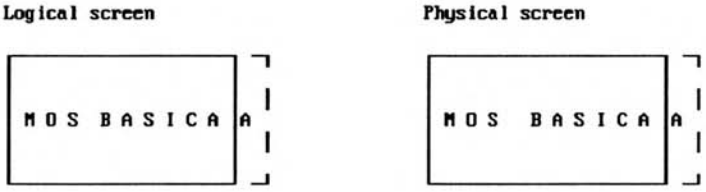

Figure 7.9c: The result of the rotation

As you can see, the new physical screen is identical to the previous version. So when we swapped between the two screens, nothing appeared to happen. If we continued this procedure, we'd find that the screen rotated at about half the original speed. We'd therefore need to perform two separate rotations for every apparent screen movement. This would generate an irritating jump in the rotation effect.

With a bit of ingenuity however, the problem can be completely sidestepped! We start off by doubling the size of our buffer zone, for reasons which should hopefully become clear in a moment. We can then scroll the logical screen exactly as before.

Now comes the clever bit. Once the screens have been initialised (Figure 7.10a), we can then simply rotate each screen by TWICE the original amount. After the rotation has concluded, we switch screens and continue. This produces the results in Figure 7.10b.

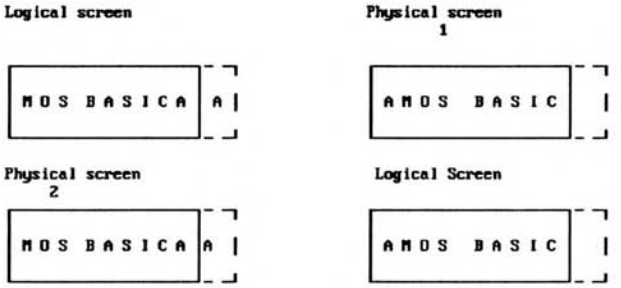

Figure 7.10a: Double buffer scrolling

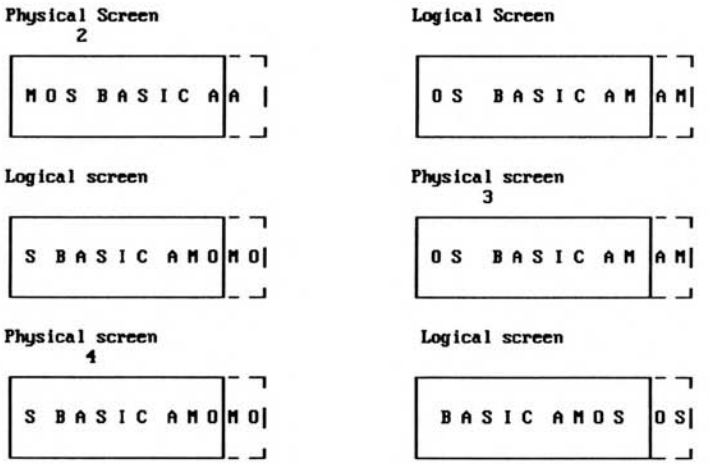

Figure 7.10b: The result after switching screens

I've numbered the physical screens from 1 to 4. If you look at each physical screen in turn, you should see that there's a smooth progression between the various images. So although we are rotating the screens by twice the usual value, each image is seemingly moved by just a single step. If you think about it, it makes a crazy sort of sense. Since we've effectively doubled the number of screens, we also need to double the size of each rotation. That's why it's called double buffering.

Example 7.3 contains a new version of our rotation program which performs all the rotation on the logical screen.

Example 7.3: Screen rotation and double buffering

```
Close Editor
Close Workbench
Rem Create screen with a 32 unit buffer area
Rem This limits you to a maximum of 16 units per each rotation
Screen Open 0,352,200,16,Lowres
Rem SCREEN DISPLAY screen_no,hx,hy,display width,display height
Screen Display 0,128,50,320,200
Double Buffer
Curs Off : Cls 0 : Paper 0 : Autoback 0 : Hide
Rem create a banner on the source screen
BANNER[" AMOS BASIC "]
Locate 0,0 : Cline
Locate 0,0 : Pen 3 : Centre "Horizontal Screen Rolling!"
Rem set up an attractive rainbow effect on the destination
Colour 2,0
Set Rainbow 1,2,32,"(2,1,16)","(2,1,16)","(2,1,16)"
Rainbow 1,0,68,256
Rem Set SPEED to twice the size of each rotation.
Rem Use values like 2,4,8,or 16
SPEED=8
Rem Switch screens to load the message into the physical screen
Screen Swap:Wait Vbl
Rem copy the message onto the new logical screen
Screen Copy Physic(0) To Logic(0)
Rem Perform a single scroll. Each rotation is SPEED/2  units long
Screen Copy Logic(0),0,50,SPEED/2,83 To Logic(0),320,50
Screen Copy Logic(0),SPEED/2,50,SPEED/2+320,83 To Logic(0),0,50
Screen Swap : Wait Vbl : Rem Flick the scrolled screen into place
Autoback 0
Rem Rotate the screen
Do
      For CS=0 To 800
         Rem scroll each screen by twice the normal amount
         Rem rotates screen from 0,50 to 320,83
         Rem change the numbers if your screen is taller or wider
         Screen Copy Logic(0),0,50,SPEED,83 To Logic(0),320,50
         Screen Copy Logic(0),SPEED,50,SPEED+320,83 To Logic(0),0,50
         Rem switch screens
         Screen Swap
         Wait Vbl
      Next CS
Loop
Procedure BANNER[A$]
  Rem This doesn't necessarily have to be a message. You could load
  rem any IFF screen into this position.
  Print A$:Zoom 0,0,0,(Len(A$)-1)*8,8 To 0,0,50,319,82
End Proc
```

Despite its complexity, the above program is really rather nice. It's certainly MUCH faster than the previous version.

There are however, a couple of definite drawbacks to this technique. As each screen needs to be displayed strictly in sequence, it's difficult to change the speed or direction of the rotation during the coarse of the program. This would require us to reset the logical and physical screens so that they were precisely in step with each other. A similar effect can be seen in certain commercial games, which pause momentarily when the scrolling direction changes.

By the way, there's nothing really stopping you from using the same procedures directly in one of your own games. The game screen can be created with Deluxe Paint, and loaded straight into memory using LOAD IFF. Providing you remember to adjust the size of your scrolling area, you can now harness my routines to rotate the entire landscape across the screen. This system was used in Peter Hickman's excellent SHOOT_EM_UP demo, supplied on early versions of the AMOS Extras disk.

But hold your fire! There's a better way!

7.3 Maps and Tiles

The problem with our current approach is that it limits us to relatively short rotation sequences. Imagine what would happen if we wanted to scroll a 100-word message through the screen. Each line of our text would need to be stored as a separate image in the Amiga's memory. Just for fun, let's have a go at working out the memory requirements of one of these super long messages.

We'll start off by making a couple of assumptions. Supposing each screen line was 320 pixels long, 32 units high, and held a maximum of 10 characters. That would represent approximately 1/6 of a screen or about 5K of memory per line. A hundred words takes up around 600 characters (including spaces), and we'd therefore need a grand total of 600/10 lines. So the total memory consumption would be:

Number of lines * 5k = (600/10)*5k = 60*5k =300k

Ouch!

The same problem would also arise in an Arcade game. We might get away with it if we kept the size of the playing area down to a couple of screens or so, but commercial games have hundreds or even thousands of different levels. How can we possibly fit these into an A500? We'd surely need megabytes of extra memory!

I'll now show you a powerful technique which can be used to generate scrolling messages of any length, without the risk of breaking the Amiga's memory bank! The same idea is equally capable of cramming hundreds of large screens into a screen scrolling game or RPG. This is the system which is used in commercial programs like Xenon II. And it's really rather easy in AMOS Basic!

7.3.1 Alphabet soup

The general idea is extremely simple. Instead of storing the entire message as a single image, we just divide it up into its component letters, and store each letter completely separately. These letters are very similar to *tiles* in a Scrabble board. Like the tiles, they can be rearranged to create any series of words or phrases you like, although there are sadly no extra bonuses for a triple word score! Another, more important difference, is that you can re-use each letter as many times as you wish. So the same set of tiles can be rearranged to generate an infinite number of possible messages.

AMOS Basic was specifically designed for games creation, and you'd naturally expect it to include several instructions for grabbing and displaying tiles. Actually Francois Lionet, the author of AMOS, got a little carried away at this point, and we eventually ended up with a total of FOUR totally independent drawing systems.

BLOCKS and CBLOCKS

These use the GET BLOCK command to grab any part of the selected screen into a special

memory area. Once it's been loaded into memory, each block can be subsequently positioned anywhere on the screen with an appropriate call to PASTE BLOCK. The standard BLOCK instructions save the area exactly as it appears on the screen. But there's also a set of CBLOCK commands which automatically compress your images before use. This saves valuable amounts of memory, although it's noticeably slower than the standard version.

In practice, BLOCKS are only useful for static backgrounds such as those found in the Magic Forest game on the AMOS Data Disk. That's because there are no facilities for storing the grabbed tile images onto the disk. You therefore need to cut the images directly off an IFF screen before running the program. This wastes valuable time, and consumes significant amounts of memory.

ICONS and BOBS

Most AMOS developers now use the more advanced ICON routines instead. These were originally intended for displaying the buttons and icons used by the AMOS Sprite Editor, but they are also ideal for the creation of our game backgrounds as well. The icon images are held in their own separate memory bank (bank 2), which is saved automatically along with our Basic programs. In order to use these icons in one of your games, you'll need to be familiar with three main commands.

`GET ICON i,tx,ty to bx,by`

GET ICON cuts a rectangular tile from *tx,ty* to *bx,by* and loads it into image number *i* in the AMOS icon bank. Allowable image numbers start from 1.

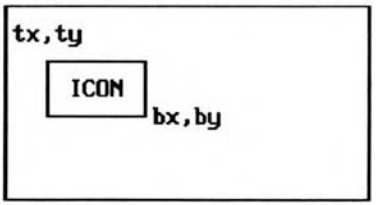

Figure 7.11: The GET ICON command

Normally, each icon will be taken straight from the current screen, but you can also specify an optional screen number in the command as well.

`GET ICON screen,i,tx,ty to bx,by`

Once you've grabbed an icon into the memory bank, you can paste it anywhere on the screen using the PASTE ICON instruction. The format is just:

`PASTE ICON x,y,i`

This pastes a copy of image number *i* on the screen, starting at coordinates *x,y*. As you would expect, the PASTE command copies the icon onto the screen using the Blitter. So all operations are performed incredibly quickly.

Warning! It's essential to make sure that the image number you are using in PASTE ICON actually exists in the current icon bank. Otherwise, AMOS can occasionally CRASH! The number of available images can be found directly using the LENGTH(2) function:

```
If I>0 and I<=Length(2) Then Paste Icon X,Y,I
```

Also note that the colours used in your icons are saved automatically in the icon bank. They can be loaded into the current screen using:

Get Icon Palette

It's obviously important to ensure that you use the same colours for your icons and Blitter objects. Incidentally, there's also a similar set of get and paste commands for Blitter objects. The format is identical to the previous icon routines, except that the images are now taken from the sprite bank (1) rather than the icon bank (2).

To grab a Bob, use:

GET BOB s,i,tx,ty to bx,by

Your sprite or Bob images can now be pasted permanently on the screen with the PASTE BOB command like so:

PASTE BOB x,y,i

Here's a small procedure which grabs an enlarged copy of the letters A to Z into icons 1 to 26.

```
Rem test routine
ALPHABET
For IC=1 to 26
Paste Icon 150,80,IC
Wait key
Next IC
Procedure ALPHABET
   Pen 2
   For C=Asc("A") To Asc("Z")
      Locate 20,0 : Print Chr$(C);
      Rem ZOOM screen1,tx1,ty1,bx1,by1 to screen2,tx2,ty2,bx2,by2
      Rem tx1,ty1,bx1,by1 specify a rectangular source area
      Rem tx2,ty2,bx2,by2 set a rectangular destination area
      Rem ZOOM enlarges or reduces the source to fit into
      Rem the destination box, and then copies the new image at
      Rem tx2,ty2. It's a nice little command
      Zoom 1,20*8,0,21*8,8 To 1,16,16,48,48
      Rem Loads "A" into icon 1, "B" into 2, etc
      Get Icon C-Asc("A")+1,16,16 To 48,48
   Next C
   Cls 0
   Rem make icon 27 into a space
   Get Icon 27,0,0 To 32,32
End Proc
```

7.3.2 Scrolling through a long message

If we want to use these tiles in our messages, we'll obviously need some way of drawing our text on the screen and then scrolling it through the display.

We'll now define two new AMOS Basic routines for just this purpose.

BANNER_DO[X]

This will draw an entire line of our message at starting from letter number *X*. We'll store our text in a simple string variable *(T$)*.

BANNER_LEFT

Let's use a GOSUB here for the maximum possible speed. The purpose of the routine will be scroll our message one place to the left, and insert a new character to the right when required as in Figure 7.12.

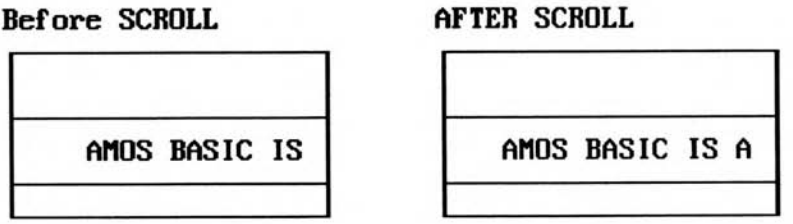

Figure 7.12: Scrolling through a message

Now you've seen the specification, here are the actual routines:

```
Procedure BANNER_DO[X]
   Rem Draw a banner starting from character number X
   Rem TX,TY = coordinates of the top corner of the scrolling zone
   Rem SWIDTH holds the width of each screen line in tiles
   Rem TWIDTH indicates the width of each individual tile
   Rem T$ holds our message
   Shared TX,TY,SWIDTH,TWIDTH,T$
   Rem Check to see if X is smaller than the length of the string
   If X>Len(T$)-SWIDTH Then Pop Proc : Bell
   Rem Display a line of text
   For C=X To SWIDTH+X
      Rem Get Cth character in the string
      P$=Mid$(T$,C,1)
      Rem convert it into a number from 1 to 26
      T=Asc(Upper$(P$))-Asc("A")+1
      Rem Check for a space
      If T=-32 Then T=27
      Rem Paste letter at appropriate screen coordinates
      If T>0 and T<=27 Then Paste Icon TX+(C-X)*TWIDTH,TY,T
   Next C
End Proc
BANNER_LEFT:
Rem S is the number of the source screen
Rem TX,TY,BX,BY holds the position of the viewing window on the source
Rem screen.
Rem SPEED is the number of points which will be moved on each scroll
Rem T$ holds the message text
Rem FX holds the viewing position
Rem TWIDTH and SWIDTH set the size of a tile and a screen line
Rem respectively.
Rem get the position of the next character in the message
TP=((FX/TWIDTH)+SWIDTH) mod Len(T$)
rem Work out screen position
SX=FX mod TWIDTH
Screen S:Rem Set PASTE operations onto SCREEN number S
rem Scroll screen S to the left
Screen Copy S,TX+SPEED,TY,BX+SPEED,BY To S,TX,TY
Rem Display the next character in our sequence
Rem equivalent to T=ASC(MID$(T$,TP,1))-ASC("A") only faster
```

```
T=Peek(Varptr(T$)+TP)-64
Rem Handle space
If T=-32 Then T=27
Rem Paste new letter on the screen
If T>0 and T<=27 Then Paste Icon BX-SX-SPEED,TY,T
Rem copy source screen to destination (0) at 0,100
Screen Copy S,TX,TY,BX,BY To 0,0,100
Rem switch screens
Screen Swap : Wait Vbl
Return
```

BANNER_DO should be pretty self-explanatory. The complicated stuff is in the BANNER_LEFT routine. If the mechanics of the system look depressingly complex, relax. All you really need to understand is how to use these routines. So feel free to skip the next section completely. Before you leave, here's some pseudo-code which shows how you can scroll a message in one of your own programs:

```
Initialise the screen
Set up double buffering (DOUBLE BUFFER)
Turn off AUTOBACK (AUTOBACK 0)
Call ALPHABET routine to create an icon for each letter
Load message text into T$
Set scrolling speed in SPEED
Display the first line of the banner with BANNER_DO[1]
Do
  Scroll the message one place to the left using Gosub BANNER_LEFT
  Move viewport to the next letter (ADD FX,SPEED)
Loop
```

FX is the X coordinate of an imaginary viewport on our message.

SPEED is the number of screen points the message is to be scrolled in each operation. The higher the value, the faster the movement!

I'll now explain the BANNER_LEFT routine an instruction at a time.

All scrolling takes place on a separate screen to avoid the screen synchronisation problems I highlighted earlier.

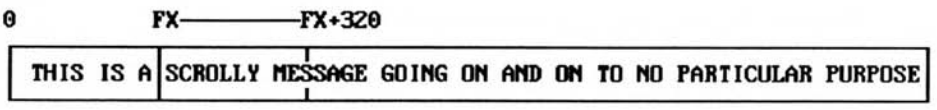

Figure 7.13: An imaginary viewport on a message

(FX/TWIDTH) holds the number of letters from the start of the message.

So *(FX/TWIDTH)+SWIDTH* represents the position of the last letter on the screen.

The *MOD Len(T$)* bit gets the remainder of this value after it's been divided by the message length. This neatly forces the entire message to repeat from the beginning when the program reaches the end of our text.
SX=FX mod TWIDTH returns the number of points in the rightmost letter on the screen. It's used to handle partly displayed letters, and allows us to scroll our message by a fraction of a single character.

We now move the entire message one place to the left using SCREEN COPY.

```
Screen Copy S,TX+SPEED,TY,BX+SPEED,BY To S,TX,TY
```

The program then works out the icon number to be displayed to the right of the screen:

```
T=Peek(Varptr(T$)+TP)-64
Rem Handle space
If T=-32 Then T=27
```

The PEEK command is just a fast way of getting the letter at position *TP* and converting it into a number from 1 to 27. It's equivalent to:

```
T=ASC(MID$(T$,TP,1))-ASC("A")
```

Once we've selected the icon, it can now be drawn at the far right of the screen using PASTE ICON:

```
Paste Icon BX-SX-SPEED,TY,T
```

Remember, *BX* is the screen coordinate of the far right of the display area, and *SPEED* is the number of points the screen will be scrolled. *SX* holds the number of points in the rightmost character which are to be displayed on the screen. If we subtract this from *BX*, we get the starting position of the new character on the screen.

Since the screen has just been scrolled however, we'll also need to subtract the speed from this value as well. The true position is given by the formula:

```
SX=BX-SX-SPEED
```

Finally, we can copy the new section of the source screen to the destination like so:

```
Screen Copy S,TX,TY,BX,BY To 0,0,100
```

We now switch screens and continue.

```
Screen Swap : Wait Vbl
Return
```

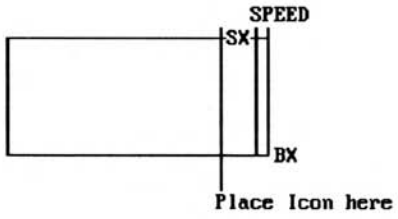

Figure 7.14

Here is a complete example of this system in action.

Example 7.4: Scrolling a message using tiles

```
Rem open viewing screen
Screen Open 0,320,200,16,Lowres
Double Buffer
Screen 0
Cls 0
Autoback 0
```

```
Rem open source screen
Screen Open 1,352,50,8,Lowres
Rem select screen one as the destination for all drawing operations
Screen 1
Rem Hide screen one away in memory
Screen Hide 1
Rem Set up screen 1
Curs Off : Cls 0 : Paper 0
Rem create alphabet
ALPHABET
Cls 0
Rem define message
T$="HI THERE THIS IS A SCROLLY MESSAGE WHICH GOES ON AND ON TO NO
PARTICULAR "
T$=T$+"PURPOSE BUT IT DOES DEMONSTRATE HOW AMOS CAN BE USED TO CREATE "
T$=T$+"ATTRACTIVE SCREEN SCROLLING EFFECTS "
SWIDTH=10
TWIDTH=32
Rem draw banner
TX=0 : TY=0 : BX=319 : BY=33
BANNER_DO[1]
Rem The speed can be one of the following values 1,2,4,8,16,32
SPEED=8
S=1 : Rem number of source screen
Rem Create a rainbow
Colour 2,0
Set Rainbow 1,2,32,"(2,1,16)","(2,1,16)","(2,1,16)"
Rainbow 1,0,150,32
Rem main loop
Do
   Gosub BANNER_LEFT
   Add FX,SPEED
Loop
Stop
Procedure ALPHABET
   Get Icon 1,0,0 To 16,16
   Make Icon Mask
   Pen 2
   For C=Asc("A") To Asc("Z")
      Locate 20,0 : Print Chr$(C);
      Zoom 1,20*8,0,21*8,8 To 1,16,16,48,48
      Get Icon C-Asc("A")+1,16,16 To 48,48
   Next C
   Cls 0
   Get Icon 27,0,0 To 32,32
End Proc
Procedure BANNER_DO[X]
   Shared TX,TY,SWIDTH,TWIDTH,T$
   If X>Len(T$)-SWIDTH Then Pop Proc : Bell
   For C=X To SWIDTH+X
      P$=Mid$(T$,C,1)
      T=Asc(Upper$(P$))-Asc("A")+1
      Rem Follow T,X,C : Follow Off
      If T=-32 Then T=27
      If T>0 and T<=27 Then Paste Icon TX+(C-X)*TWIDTH,TY,T
   Next C
End Proc
BANNER_LEFT:
Screen S
TP=((FX/TWIDTH)+SWIDTH) mod Len(T$)
```

```
SX=FX mod TWIDTH
Screen Copy S,TX+SPEED,TY,BX+SPEED,BY To S,TX,TY
T=Peek(Varptr(T$)+TP)-64
If T=-32 Then T=27
If T>0 and T<=27 Then Paste Icon BX-SX-SPEED,TY,T
Screen Copy S,TX,TY,BX,BY To 0,0,100
Screen Swap : Wait Vbl
Return
```

In our previous scroller, it was possible to speed up the entire scrolling operation by performing all our commands on the invisible logical screen. The general procedure was as follows:

```
Load the physical and logical screens with our message (or landscape)
Scroll the logical screen by a single step
Switch screens to display the results in the physical screen
Do
  Scroll the logical screen by TWICE the original step size
  Swap between the logical and physical screens
Loop
```

I'll now provide you with a brand new version of Example 7.4 to demonstrates this technique.

Example 7.5: Fast message scroller

```
Rem This is almost identical to 7.4
Rem So you won't need to retype the whole thing again!
Rem Open screen with an extra 32 points for invisible scroll buffer
Screen Open 0,352,200,16,Lowres
Double Buffer
Rem Limit the width of the screen to 320 characters
Screen Display 0,128,,320,
Rem turn off automatic drawing system
Autoback 0
Curs Off : Cls 0 : Paper 0
Rem Make all the icons
ALPHABET
Rem Create a rainbow
Colour 2,0
Set Rainbow 1,2,50,"(2,1,16)","(2,1,16)","(2,1,16)"
Rainbow 1,0,50,255
Rem Define message. This can be ANYTHING you like. But it must be
Rem in CAPITAL LETTERS.
T$="THIS IS A TEST OF ULTRA SMOOTH HORIZONTAL SCROLLING IN AMOS BASIC "
T$=T$+"CARE OF THE GAME MAKERS MANUAL BY STEPHEN HILL"
Rem Define WIDTH of SCROLLING AREA in tiles
SWIDTH=10
Rem Set the width of each tile
TWIDTH=32
Rem Set the position of the scrolling area. Change at will
TX=0 : TY=100 : BX=320 : BY=133
Rem Draw the first bit of the message
BANNER_DO[1]
Rem load the message into both the physical and logical screens
Screen Swap : Wait Vbl
Screen Copy Physic(0) To Logic(0)
Rem allowable speeds are 1,2,4,8,16
Rem Scroll the message once to set up the sequence
SPEED=2 : Gosub BANNER_LSET
FX=SPEED
```

```
Rem Perform each subsequent scroll by TWICE the normal amount
SPEED=SPEED*2
Rem Main loop
Do
   Rem Scroll screen one place to the left
   Gosub BANNER_LEFT
   Rem Move viewing area one place to the RIGHT
   Add FX,SPEED/2
Loop
Stop
Procedure ALPHABET
   Get Icon 1,0,0 To 16,16
   Make Icon Mask
   Pen 2
   For C=Asc("A") To Asc("Z")
      Locate 20,0 : Print Chr$(C);
      Zoom 0,20*8,0,21*8,8 To 0,16,16,48,48
      Get Icon C-Asc("A")+1,16,16 To 48,48
   Next C
   Cls 0
   Get Icon 27,0,0 To 32,32
End Proc
Procedure BANNER_DO[X]
   Shared TX,TY,SWIDTH,TWIDTH,T$
   If X>Len(T$)-SWIDTH Then Pop Proc : Bell
   For C=X To SWIDTH+X
      P$=Mid$(T$,C,1)
      T=Asc(Upper$(P$))-Asc("A")+1
      If T=-32 Then T=27
      If T>0 and T<=27 Then Paste Icon TX+(C-X)*TWIDTH,TY,T
   Next C
End Proc
BANNER_LEFT:
TP=((FX/TWIDTH)+SWIDTH) mod Len(T$)
SX=FX mod TWIDTH
Screen Copy Logic(0),TX+SPEED,TY,BX+SPEED,BY To Logic(0),TX,TY
T=Peek(Varptr(T$)+TP)-64
If T=-32 : T=27 : End If
If T>0 and T<=27 : Paste Icon BX-SX-SPEED/2,TY,T : End If
Screen Swap : Wait Vbl
Return
BANNER_LSET:
TP=((FX/TWIDTH)+SWIDTH) mod Len(T$)
SX=FX mod TWIDTH
Screen Copy Logic(0),TX+SPEED,TY,BX,BY To Logic(0),TX,TY
T=Peek(Varptr(T$)+TP)-64
If T=-32 : T=27 : End If
If T>0 and T<=27 : Paste Icon BX-SPEED,TY,T : End If
Screen Swap : Wait Vbl
Return
```

Example 7.5 is practically identical to the original version. The only real difference is the PASTE ICON command in the BANNER_LEFT routine, which positions the letter slightly further to the right using a value of SPEED/2 rather than the original SX. There's also a new BANNER_LSET routine which sets up the banner during initialisation. Naturally, I could have extended BANNER_LEFT to perform both operations, but it was simpler to use a separate procedure for this purpose instead.

7.3.3 The AMOS MAP definer

This is where the fun really starts! I'll now show you how the same techniques can be used to generate the massive scrolling playfields required by a real game. The playfield will be created out of number of building blocks or tiles. Each tile holds a small section of the final image, such as a piece of wall, or part of the screen background. By re-arranging our tiles in a particular order, we can generate thousands of different screens out of a relatively small number of icons.

But how can we actually create one of these maps? Well, AMOS Basic provides us with a crude, but useful map editor called TAME on the AMOS program disk. The tiles are taken from a large IFF screen which must be created in advance with an external drawing package such as Deluxe Paint. This file is then loaded into TAME, and split into its component building blocks. You can now draw your map directly on the screen using the mouse.

Originally, the tiles were grabbed into memory using the AMOS Basic GET BLOCK commands. They could then be *pasted* onto the appropriate screen position using a simple PASTE BLOCK instruction. Experience has shown that it's better to store our images as icons instead. These images are stored in their own separate memory bank, and can be saved automatically along with our Basic programs. Here's a small *tile cutter* which converts an IFF picture into an icon bank.

```
Rem Tile cutter
Input "Enter Tile width  ?";W
Input "Enter Tile height ?";H
Do
   Rem Load up a tile screen in IFF format
   F$=Fsel$("","","Load an IFF file")
   Exit If(F$="")
   Exit If Exist(F$)=0
   Load Iff F$,0
   Gosub TILE_CUTTER
   Rem Draw screen using new tiles
   Cls 0 : Wait 10 : IC=1
   For Y=0 To SH-1
      For X=0 To SW-1
         Paste Icon X*W,Y*H,IC : Inc IC
      Next X
   Next Y
   Rem Save tiles in a memory bank
   F$=Fsel$("*.abk","","Save an Icon file")
   Exit If F$=""
   Save F$,2
Loop
Stop
TILE_CUTTER:
Erase 2
Rem Chop an IFF screen into its component tile icons
SW=Screen Width(S)/W : SH=Screen Height(S)/H : IC=1
For Y=0 To SH-1
   For X=0 To SW-1
      Get Icon IC,X*W,Y*H To(X+1)*W,(Y+1)*H
      Inc IC
    Next X
Next Y
Return
```

7.4 AMOS TOME (TOtal Map Editor)

If you're intending to draw a lot of maps, you'll probably find it easier to junk TAME completely, and invest in the extended AMOS TOME system. This is available from the AMOS user group for £24.95 (£20 for club members), and provides you with everything you need to create large game maps with the absolute minimum of effort.

TOME is published by a small, inconspicuous little company called Shadow Software. It's run by Aaron Fothergill from the AMOS User Group, along with his brother Adam. Aaron handles the programming side, and Adam's responsible for designing the various graphics. The results of this collaboration can be seen in the excellent Magic Forest game on the AMOS Data Disk. They've also produced several full length budget games in AMOS Basic!

AMOS TOME is a new improved version of the original AMOS MAP definer. If you hated the original version, you'll be happy to hear that it's been given a TOTAL rewrite, and now bears no real resemblance to TAME. Note: AMOS TOME is a large (130K) AMOS program, and therefore requires at least a megabyte of memory. Hopefully, this shouldn't be a problem, as you'll probably have upgraded ages ago. If not, now's the time. The extra memory will come in very handy when you're developing your screen scrolling games!

The entire TOME system is controlled through a number of screen icons or buttons which can be selected with the mouse in the normal way. The editor includes all the facilities you'd expect from a standard drawing package, such as cut and paste, block fills, etc. However, instead of working in single screen points, all operations are performed in units of a complete tile. This certainly takes some getting used to, but after you've been using the system for a while, most operations quickly become second nature.

TOME supports a range of advanced facilities such as automatic map generation, and a tile valuer which allows you to assign up to four numerical values to each individual icon. My favourite of these features is the icon editor, which lets you make up a tile directly on the screen, without having to continually jump back and forth to an external drawing package such as Deluxe Paint or Spritex.

The latest version also includes a neat overview command, which displays a reduced image of the entire map on the screen. This is really invaluable when you're designing a game, as it enables you to take in the whole playfield in a single glance.

Maps are saved in a standard format, and can be loaded straight from the original TAME program. You do however, need to convert your old tile images from an IFF screen file into icons. This can be achieved with the aid of an extended Tile Maker utility supplied on the TOME disk. One word of warning though. Don't attempt to display TOME maps with the Map View routine on the AMOS Program disk. Due to a slight difference in the map formats, Map View will immediately crash your machine!

In addition to the Map Editor, you also get a number of powerful extensions to the AMOS system. These add 32 extra AMOS commands which can be used to display and scroll your maps directly from AMOS Basic. Once they've been installed onto your AMOS Program disk, they are permanently on tap, and can be compiled and run just like any other Basic instruction.

The AMOS compiler requires a separate extension file in order to compile TOME programs. Shadow Software will happily supply a free upgrade to the latest system to all registered TOME users. So if you've forgotten, it might be a good time to send off your TOME registration card. By the way, unlike the TOME Editor, the extension routines work fine on ANY Amiga. Memory isn't a problem, since they're only about 3.5K long!

Just to whet your appetite, here's a quick selection of the more important instructions. The full system includes an amazing 32 commands. If you don't have a copy of TOME, relax. I'll be

also providing you with a useful set of TOME simulation routines which mimic many of these commands from standard AMOS Basic. So you won't have to rush out immediately and buy a copy of TOME in order to run my examples. Sorry Aaron!

7.4.1 The AMOS TOME extension

MAP VIEW tx,ty TO bx,by
MAP VIEW chooses the position and size of the TOME map display relative to the current screen. *tx,ty* set the coordinates of the top left hand corner of the map drawing area, and *bx,by* hold the coordinates of the point diagonally opposite. As a default, the map will be drawn on the current screen from 0,0 to 320,200.

MAP DO mx,my
This displays a section of the map starting at tile coordinates *mx,my*. The map is automatically clipped to fit into the previously defined viewing area. Tile coordinates are measured in units of a single icon. So *mx* holds the number of tiles horizontally from the left of the map, and *my* holds the number of tiles down from the top.

=MAP X

=MAP Y
These return the width and height of the map in tiles.

MAP LEFT mx,my

MAP RIGHT mx,my

MAP TOP mx,my

MAP BOTTOM mx,my

If you think back to the previous section on message scrolling, you may recall that the BANNER_LEFT command performed two totally separate operations. It first copied the message one place to the left using SCREEN COPY, and then redrew the final letter at its new position with PASTE ICON.

The same idea can also be applied to TOME maps. In this case, we'll need to draw a whole LINE of tiles rather than just a single letter. TOME includes four special commands for this purpose, shown in Figure 7.15.

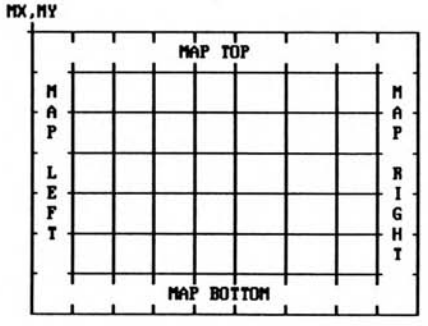

Figure 7.15: The TOME map commands

MAP LEFT mx,my
MAP LEFT draws the line of tiles starting from the left hand side of the current map. *mx,my* hold the map coordinates of the top left corner of the map.

MAP RIGHT mx,my
Draws a line of tiles forming the right hand border of the map. The tiles are taken from the map coordinates *mx+window_width,my* where *window_width* represents the width of the current drawing area in tiles.

MAP TOP mx,my
MAP TOP redraws the line of tiles along the top of the display.

MAP BOTTOM mx,my
This displays the tiles along the bottom row of the current display window.

In a short while, I'll be showing you how you can use these instructions to scroll a real map onto the screen.

TILE SIZE w,h
TILE SIZE informs TOME of the width and height of the icons you'll be using for your tiles. This command is essential, and should be executed as part your program's initialisation phase. Allowable values for *w* and *h* range from 1 to 32, but usually you'll want to use values such as 16 or 32.

MAP BANK b
Usually, TOME expects to find the map data in memory bank 6. This can be changed to any bank you like, allowing you to use several maps in the same program.

MAP PLOT t,mx,my
MAP PLOT replaces the tile at map coordinates *mx,my* with tile number *t*. Note that this has NO effect on the screen. The new tile will only be displayed when the map is redrawn.

t=MAP TILE(MX,MY)
MAP TILE returns the number of the current tile at map coordinates *mx,my*.

v=TILE VAL(mx,my,l)
This checks the tile at coordinates *mx,my* and returns a value you've previously assigned using the tile valuer. The tile valuer holds a list of up to four separate numbers for each individual icon. *l* indicates the list number you wish to check ranging from 0 to 3.

TILE BANK b
Changes the number of the memory bank used to hold the tile values.

PASTE BRIK n,x,y
This pastes TOME brik number *n* onto the screen at coordinates *x,y*. Briks are rectangular groups of tiles which can be created with the TOME map editor. You can think of Briks in the following way: *Words are to letters what briks are to tiles.*

Briks are perfect for large background areas or the walls of a Dungeon Master type game. Note: PASTE BRIK only draws a brik on the screen. It has absolutely no effect on the contents of your actual map.

The final component of the AMOS TOME package is the example programs. These give an impressive demonstration of the system in action, and provide dozens of valuable pointers to the creation of your own AMOS Basic games. Overall, I can heartily recommend the AMOS TOME system to any serious AMOS user.

Having said that, it's important to recognise that TOME is just a specialised version of a standard drawing package. Like all such utilities, the results vary in direct proportion to the effort you are prepared to put into using it. If you want to generate commercial quality game

screens, you will need to spend a great deal of time drawing the tiles and designing the various maps. Although TOME can simplify the process enormously, it still requires a considerable amount of work on the part of the user. So it's unreasonable to expect TOME to do the whole job for you!

7.4.2 Displaying a TOME map

On paper, displaying a TOME map is just a simple matter of defining the map window with MAP VIEW and typing MAP DO. As usual though, it's not quite that simple. Before we can draw our map on the screen, we first need to indulge in some careful initialisation work. Here's a step by step guide to displaying a TOME (or TAME) map on the screen.

- First, open the AMOS screen which will be used to hold your TOME maps. If you're subsequently intending to scroll your map, you'll usually need to define an invisible background screen for this purpose. This will allow you to draw your maps behind the scenes, without interfering with your current program display. Example:

```
Screen open 1,320,200,16,Lowres
Screen Hide 1:Rem Hide screen 1 away from view
```

- Next, inform TOME of the width and height of the icons you are using to construct the map. Use a command like:

```
Tile Size 16,16
```
(or `TWIDTH=16:THEIGHT=16` using the TOME Simulator)

- Now set the position and size of the drawing area used to create your maps. This can be achieved with the MAP VIEW command like so:

```
TX=0:TY=0:BX=320:BY=200: Map View TX,TY to BX,BY
TX=0:TY=0:BX=320:BY=200: Gosub TVIEW:Rem   from the simulator
```

- You can then load the various icon images you'll be using for your tiles.

```
Load "filename.abk"
```

- Next, reserve some space for your MAP data in memory bank six. The size of this bank depends on the number of rows and the number of columns in your game map. It's given by the formula:

  Memory=Rows*Columns+4

A typical line would be:

```
Reserve As Data 6,40*25+4
```

- The map data can now be loaded from the disk into the newly created bank with a line such as:

```
Bload "mapname.map",Start(6)
```

- If you're using the optional tile valuer, you'll also need to load your tile values into memory. For example:

```
Reserve as Work 8,1024:Rem The tile valuer uses a max of 1k
Bload "mapname.map.val",Start(6)
```

- Finally, you can read the colours used by the icons into the screen with:

```
Get Icon Palette
```

That's it. You can now display your map with a simple call to Map Do:

Example 7.6

```
Rem Set the width and height of your tiles
TWIDTH=16 : THEIGHT=16
Rem Tile Size TWIDTH,THEIGHT for TOME USERS
Rem Make some icons
MAKE_ICONS
Rem Set up screen
Double Buffer
Rem load map
Gosub MAP_INIT
Rem set position of viewing area on map
TX=80 : TY=32 : BX=240 : BY=160 : Gosub TVIEW
Rem TOME USERS should  add a REM before GOSUB TVIEW and activate
Rem the next four lines
Rem Map View TX,TY To BX,BY
Rem W=Abs(BX-TX) : H=Abs(BY-TY)
Rem SWIDTH=W/TWIDTH : If W mod TWIDTH<>0 Then Inc SWIDTH
Rem SHEIGHT=H/THEIGHT : If H mod THEIGHT<>0 Then Inc SHEIGHT
Cls 0 : Locate 0,0 : Centre "<A simple TOME MAP>"
Autoback 0 : Rem Turn off AUTOBACK system
Rem Draw first section of the map
MX=0 : MY=0 : Gosub TDO : Rem Map Do MX,MY for TOME
Rem Copy into both physical and logical screens
Screen Swap : Wait Vbl
Screen Copy Physic To Logic
Rem Main loop
Do
   OX=MX : OY=MY
   Rem Move through map using the joystick
   If Jleft(1) and MX>0 Then Dec MX
   If Jright(1) and MX<MPX-SWIDTH Then Inc MX
   If Jup(1) and MY>0 Then Dec MY
   If Jdown(1) and MY<MPY-SHEIGHT+1 Then Inc MY
   If OX<>MX or OY<>MY Then Gosub TDO : Screen Swap : Wait Vbl
   Rem TOME users should replace the previous line with
   Rem If OX<>MX or OY<>MY Then Map Do MX,MY : Screen Swap : Wait Vbl
Loop
Rem Here's TINY_TAME
Rem It takes a string containing the numbers 1-9, the letters A-Z or a
Rem space  and converts it into a TOME MAP
Rem Space is icon 1. A=10,B=11 and so on until Z=36
Procedure TINY_TAME[T$,W,H]
   Rem reserve data area
   Erase 6 : Reserve As Data 6,W*H+50
   Rem store width and height
   Doke Start(6),W : Doke Start(6)+2,H
   Rem load map
   For M=0 To Len(T$)-1
      Rem get Mth character in T$
      I1=Peek(Varptr(T$)+M)
      Rem space=icon 1
      If I1=32 Then IC=0
      Rem handle numbers (All TILE values  are  ONE LESS than the icon NO
      If I1>=Asc("1") and I1<=Asc("9") Then IC=I1-Asc("1")
      Rem handle letters (A=10,B=11..etc)
      If I1>=Asc("A") and I1<=Asc("Z") Then IC=I1-Asc("A")+10
      Rem Lowercase letters are treated the same as capitals
      If I1>=Asc("a") and I1<=Asc("a") Then IC=I1-Asc("a")+10
```

```
      Rem check for an error
      If IC>Length(2) or IC<0 Then IC=0
      Rem store tile number
      Poke Start(6)+M+4,IC
   Next M
End Proc
Procedure MAKE_ICONS
   Rem Since this is a listing, I'm having to make up the icons
   Rem as I go along...
   Curs Off : Cls 0 : Paper 0
   Locate 0,20 : Centre "<Please wait>"
   Pen 15
   Rem first icon is a blank space
   Get Icon 1,0,0 To 16,16
   Rem make some blocks
   For B=2 To 8
      Ink B+4
      Bar 0,0 To 15,15
      Ink 2 : Box 0,0 To 15,15
      Get Icon B,0,0 To 15,15
      Cls 0,0,0 To 17,17
   Next B
End Proc
TVIEW:
Rem This routine sets up the window for TOME
S6=Start(6) : MPX=Deek(S6) : MPY=Deek(S6+2)
TWIDTH=Max(16,TWIDTH) : THEIGHT=Max(16,THEIGHT)
W=Abs(BX-TX) : H=Abs(BY-TY)
SWIDTH=W/TWIDTH : If W mod TWIDTH<>0 Then Inc SWIDTH
SHEIGHT=H/THEIGHT : If H mod THEIGHT<>0 Then Inc SHEIGHT
Return
TDO:
Rem This routine draws the map in the window set by the TVIEW routine
Rem at map co-ordinates MX,MY
S6=Start(6)+4
YS=MY*MPX
Rem The next bit's vital, as AMOS hates PASTING nonexistent icons
Rem So if you try to print off the edge of the map, AMOS can CRASH!
If MX<0 or MY<0 or MX>MPX-SWIDTH or MY>MPY-SHEIGHT+1 Then Boom : Return
For Y=0 To SHEIGHT-1
   XS=MX
   For X=0 To SWIDTH-1
      BL=Peek(S6+(XS mod MPX)+YS)
      Paste Icon TX+X*TWIDTH,TY+Y*THEIGHT,1+BL
      Inc XS
   Next X
   Add YS,MPX
Next Y
Return
MAP_INIT:
Rem Define map (using the new, amazing TINY_TAME map editor)
T$=""
Rem The rem with the numbers is a RULER. It's exactly 60 columns long.
Rem Place your finger on the zero key, and press the numbers 0-9, six
Rem times in succession. Start at column 8 and end at column 67.
Rem There are 24 lines of data to enter!
Rem Be careful to keep the text lined up
Rem An extra space in the wrong position can muck up the entire map
Rem     012345678901234567890123456789012345678901234567890123456789
T$=T$+"2222222222222222222222222222222222222222222222222222222222222222"
T$=T$+"2                                                            2"
```

```
T$=T$+"23333333   3333333333333333333333333333333333333333333333  2"
T$=T$+"2      3   4          5      6                            3  2"
T$=T$+"2      3   4 44444    5      6666666666    666666666666  3  2"
T$=T$+"2          4 4   4    5            66      66    6   6  3  2"
T$=T$+"2      3   4 444 4    5      66666666        6        6     2"
T$=T$+"2      3   4     4    5                              6   6  3  2"
T$=T$+"23333333   4444444    5      66666666666666666666666666  3  2"
T$=T$+"2    3                                    6   6            3  2"
T$=T$+"2    333333333333333333333333333      6   6    3333333333  2"
T$=T$+"2          4                            6   6               2"
T$=T$+"2          4                            6   6               2"
T$=T$+"2          4                                                2"
T$=T$+"233333333333333  33333333333333333333333333333333333333  2"
T$=T$+"2          4        4                                    3  2"
T$=T$+"2          4        4        444444444          44444444  3  2"
T$=T$+"2          4        4        4                        4  3  2"
T$=T$+"2          4        4        4     2222222222222222   4  3  2"
T$=T$+"2          4        4        4                        4  3  2"
T$=T$+"23333333   444444444 5  4444444                   5555  3  2"
T$=T$+"2                   5                                    3  2"
T$=T$+"2    333333333333333333333333333333333333333333333333333  2"
T$=T$+"2    3                                                    2"
T$=T$+"22222222222222222222222222222222222222222222222222222222222"
Rem Make map using the TINY_TAME map definer
Rem The second half is only the same to reduce the amount of typing
Rem Providing the size remains unchanged, you can use anything you
Rem like
TINY_TAME[T$,60,24]
Return
```

Obviously Example 7.6 requires some explanation. It includes a selection of the TOME simulation routines I promised you earlier, and also introduces you to the TINY_TAME Map definer. Here's a detailed breakdown of the various routines.

TVIEW

This is very similar to the TOME MAP VIEW instruction. It sets the size and position of the viewport which will be used to display your TOME maps.

TX,TY hold the screen coordinates of the top left hand corner of the current viewing area. *BX,BY* set the coordinates of point diagonally opposite.

After calling TVIEW, the following four variables will be defined.

SWIDTH will be loaded with the number of tiles which will fit in each line of the viewport. *SHEIGHT* will store the height of the window in tiles. *MPX,MPY* contain the width and height of the entire map in tiles. They're equivalent to the MAP X and MAP Y functions from AMOS TOME.

TDO

Draws the selected area of the map in the previously defined viewport.

MX,MY hold the TILE coordinates of the section of the map you wish to view. TDO has exactly the same effect as the TOME MAP DO command. However, since MAP DO is written in pure machine code it's considerably faster! Technically minded readers may like to note that the map is stored in the following format:

0: Width of map in tiles

2: Height of map in tiles

4: Start of map data

The data itself is just a list of the tile numbers in each map position. Each position is held in a single byte, and can therefore take value ranging from 0-255. Since icon numbers start at one, TOME automatically subtracts this from the icon number before saving the data into memory. So the tile numbers are exactly one less than the icons they represent. In other words, Icon 1 is stored as Tile 0, Icon 2 as Tile 1, etc. The tiles are arranged in the order they will be displayed on the screen:

```
 1, 2, 3, 4, 5, 6, 7          First line
 8, 9, 10, 11, 12, 13, 14     Second line
15, 16, 17, 18, 19, 20, 21    Third line
```

and so on. TAME maps are very similar, except that there's some extra (and fairly useless) data tacked on the end of the tile positions. If you ignore this data, you can display both TAME and TOME maps using the same general purpose routines.

TINY_TAME[T$,W,H]

TINY_TAME is a simple map definer which allows you to enter your maps directly into AMOS Basic. The map definition is contained from a standard AMOS string variable which can be edited from the Amos Editor window. W,H hold the width and height of the map in tiles.

Each character in *T$* represents a single tile in the new map. It's loaded with the number of the icon image which is to be stored at the current position. The present system provides you with a maximum of 36 different tiles per map, but this could be easily extended if required.

The first nine icons are allocated to the equivalent numbers (1 to 9). As tile number one is generally used for the background areas of your map, I've also assigned it to the space character. This makes the final map definition a great deal easier to read. The icons from 10 to 36 are entered using the letters A to Z. So A refers to icon number 10, B to icon 11 and so on.

Note that there's absolutely NO difference between lowercase letters and capitals. TINY_TAME therefore converts the letters A and a to exactly the SAME number (10). The map is entered exactly as it will appear on the screen, one line at a time.

```
T$=""
T$=T$+"33333333333333333333":Rem Line 1
T$=T$+"2                   2":Rem Line 2
T$=T$+"33333333333333333333":Rem Line 3
TINY_TAME[20,3]
```

This generates a simple 20 by 3 box composed of tile numbers 2 and 3. TINY_TAME is not, of coarse, intended as a replacement for the TAME or TOME map editors. That would be insane! It does however, provide a fast method of entering a map from the printed page, without resorting to tedious lists of data statements. Feel free to use it in any of your own listings, even if intended for publication. As long as you include a quick mention of the Game Maker's manual in your program, you are welcome to borrow it without any restrictions. Ok?

7.4.3 Scrolling a TOME map

The best thing you can say about the previous demo was that it more or less worked. Not only was it extremely slow, but the movement jumped in 16 unit increments. As it stands, it's totally unusable in a real game. So you're probably expecting me to come up with a cunning plan of some sort. Well I've some bad news for you.

There are actually FOUR possible solutions! Don't panic however! Most of these techniques should be pretty familiar, as they are just expanded versions of the text scrollers we looked at earlier. Each system has its own unique set of advantages and disadvantages.

I'll begin by introducing you to a few general terms:

The *map buffer* refers to the AMOS Basic screen you'll be using to actually draw your maps. This could be either the logical part of the main display area, or a totally separate screen. Either way, the map buffer will be hidden from view during the entire scrolling operation.

The *viewport* holds the section of the map which is physically displayed on your TV set. This viewport usually will be double buffered, and will therefore have both a logical and physical component. Depending on the circumstances, the size of the viewport could be VERY different to that of the map buffer.

The *playfield* is the technical name for the imaginary map display we've created using our Map Editor. It's this display which we will be scrolling through the viewport. As I mentioned previously, the playfield is stored as a list of its component tiles in one of the AMOS memory banks. It can be displayed on the screen, a section at a time using the MAP DO command or the TDO routine. So the playfield is not an actual image at all, merely a potential one, waiting to be drawn.

Coarse scrolling

If you're writing an RPG, you may be perfectly happy to scroll your playing area in units of a single tile. Providing your party can move across the landscape at an acceptable speed, the quality of the scrolling effects may be largely irrelevant.

In this case, you can use the following scrolling system: First draw up the map using the MAP DO command as usual. Whenever the party moves, the playfield should be scrolled in the current movement direction, and the next part of the map displayed at the edge of the screen. Instead of redrawing the entire map buffer at this point, you just scroll it in the required direction, and draw a new line of tiles at the appropriate screen boundary as shown in Figure 7.16. The new tiles can be drawn at the edge of the screen using one of the following four TOME commands:

`Map Top`

`Map Bottom`

`Map Left`

`Map Right`

Supposing you wished to scroll your map on screen number *S*. You could now use the following code:

Scroll left

```
Screen Copy S,TX,TY,BX-TWIDTH,BY To S,TX+TWIDTH,TY
Screen S : Map Left MX,MY:Rem or Gosub TLEFT
```

Remember, *TX,TY,BX,BY* hold the coordinates of the area to be copied from the Map Buffer (screen *S*), and *TWIDTH,THEIGHT* set the size of each tile in pixels (or screen points).

Scroll right

```
Screen Copy S,TX+TWIDTH,TY,BX,BY To S,TX,TY
Screen S : Map Right MX,MY: Rem or Gosub TRIGHT
```

Scroll up

```
Screen Copy S,TX,TY,BX,BY-THEIGHT To S,TX,TY+THEIGHT
Screen S : Map Top MX,MY:Rem Gosub TTOP
```

Scroll down

```
Screen Copy S,TX,TY+THEIGHT,BX,BY To S,TX,TY
Screen S : Map Bottom MX,MY:Rem or Gosub TBOTTOM
```

Once you've scrolled your map buffer in this way, you can then copy it straight into the viewport using a command like:

```
Screen Copy S,TX,TY,BX,BY To Logic(0),TX,TY
```

See how I've copied the map buffer into the logical part of the viewport. After the screen copy command has done its work, the logical screen can be flicked into place with a simple SCREEN SWAP command, and the new section of the map will immediately displayed.

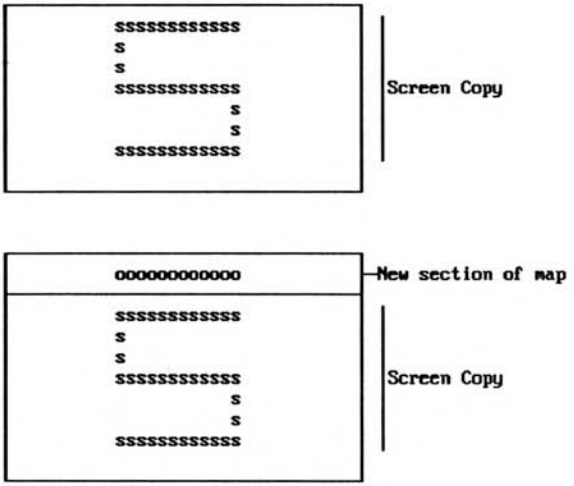

Figure 7.16: Scrolling the screen

Example 7.7: Coarse scroller

```
Rem This example will NOT run on its own.
Rem You'll need to add the routines
Rem MAKE_ICONS,TVIEW,TDO,MAP_INIT and TINY_TAME from Example 7.6
Rem Place them at the end of the listing. OK?
Double Buffer
Curs Off : Cls 0 : Centre "<Please wait>"
TWIDTH=16 : THEIGHT=16
Rem Tile Size TWIDTH,THEIGHT for TOME USERS
Rem open a screen for our map buffer
Screen Open 1,320,200,16,Lowres
Rem Screen 1 is current, and will be used for all future
Rem graphics operations
Rem Hide the screen used for map buffer completely from view
Screen Hide 1
```

```
Rem create tiles
Rem Insert original MAKE_ICONS routine at end
MAKE_ICONS
Rem set up map
Gosub MAP_INIT : Rem MAP_INIT is NOT included in this listing
Rem Define View Port
TX=80 : TY=32 : BX=240 : BY=160 : Gosub TVIEW
Rem TOME USERS should  add a REM before GOSUB TVIEW and activate
Rem the next four lines
Rem Map View TX,TY To BX,BY
Rem W=Abs(BX-TX) : H=Abs(BY-TY)
Rem SWIDTH=W/TWIDTH : If W mod TWIDTH<>0 Then Inc SWIDTH
Rem SHEIGHT=H/THEIGHT : If H mod THEIGHT<>0 Then Inc SHEIGHT
Rem Add a title
Curs Off : Cls 0 : Locate 0,0 : Pen 2 : Paper 0 : Centre "<A simple TOME MAP>"
Rem Draw our Map on screen 1
MX=0 : MY=0 : Gosub TDO : Rem Map Do MX,MY for TOME
Screen 0
Rem Draw a box around the map
Ink 1 : Box TX-1,TY-1 To BX+1,BY+1
Rem copy map screen into screen 0
Screen Copy 1 To Logic(0)
Rem Make sure that Logical and physic screen contain identical images
Screen Swap
Screen Copy 1 To Logic(0)
Pen 2 : Paper 0
Autoback 0
Rem Screen Show 1
Rem Main loop
Do
   OX=MX : OY=MY
   Rem Scroll left
   If Jleft(1) and MX>0
      Dec MX
      Screen Copy 1,TX,TY,BX-TWIDTH,BY To 1,TX+TWIDTH,TY
      Screen 1 : Gosub TLEFT : Rem or Map Left MX,MY
   End If
   Rem Scroll right
   Rem SWIDTH=no of tiles in each line of the viewing area
   If Jright(1) and MX<MPX-SWIDTH
      Inc MX
      Screen Copy 1,TX+TWIDTH,TY,BX,BY To 1,TX,TY
      Screen 1 : Gosub TRIGHT : Rem or Map Right MX,MY
   End If
   Rem Scroll up
   If Jup(1) and MY>0
      Dec MY
      Screen Copy 1,TX,TY,BX,BY-THEIGHT To 1,TX,TY+THEIGHT
      Screen 1 : Gosub TTOP : Rem Map Top MX,MY
   End If
   Rem Scroll down
   Rem SHEIGHT=No of tiles in each column of the viewing area
   If Jdown(1) and MY<MPY-SHEIGHT+1
      Inc MY
      Screen Copy 1,TX,TY+THEIGHT,BX,BY To 1,TX,TY
      Screen 1 : Gosub TBOTTOM : Rem Map Bottom MX,MY
   End If
   If OX<>MX or OY<>MY
      Boom
      Screen 0 : Screen Copy 1,TX,TY,BX,BY To Logic(0),TX,TY
```

```
      Locate 0,2 : Cline : Print "Map coords",MX,MY
      Screen Swap : Wait Vbl
   End If
Loop
Rem TOME SIMULATORS
Rem Map Left
TLEFT:
S6=Start(6)+4
YS=MY*MPX
XS=MX
For Y=0 To SHEIGHT-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon TX,TY+Y*THEIGHT,1+BL
   Add YS,MPX
Next Y
Return
TRIGHT:
Rem Map Right
S6=Start(6)+4
YS=MY*MPX
XS=MX+SWIDTH-1
For Y=0 To SHEIGHT-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon BX-TWIDTH,TY+Y*THEIGHT,1+BL
   Add YS,MPX
Next Y
Return
TTOP:
Rem Map Top
S6=Start(6)+4
YS=MY*MPX
XS=MX
For X=0 To SWIDTH-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon TX+X*TWIDTH,TY,1+BL
   Inc XS
Next X
Return
TBOTTOM:
Rem Map Bottom
S6=Start(6)+4
YS=(MY+SHEIGHT-1)*MPX
XS=MX
For X=0 To SWIDTH-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon TX+X*TWIDTH,BY-THEIGHT,1+BL
   Inc XS
Next X
Return
Rem Add TINY_TAME,TDO,TVIEW,MAP_INIT and MAKE_ICONS from Example 7.6
```

Here's a summary of the pros and cons of this system.

Advantages

❑ Fast, easy to program, and uses very little memory.

Disadvantages

❑ The scrolling effect is quite jerky, so it's useless for an arcade game.

Fine scrolling through a large map buffer

If you've added extra memory to your Amiga, you may be able to expand your invisible map buffer to include several whole screenfuls of the playing area at a time. With the help of the SCREEN OPEN command, you can safely generate screens up to about 800 points wide by 800 high. This would allow you to move through the map buffer using a simple version of my original message scroller. After you'd defined your screen, you'd load it with the first section of the game map with MAP DO (or TDO). As shown in Figure 7.17 you could then move the viewport through the current display, using a single SCREEN COPY command like so:

```
Screen Copy 1,IX,0,IX+SWIDTH,SHEIGHT To Logic(0),STX,STY
```

IX is the X coordinate of the section of the map buffer to be displayed. *SWIDTH* and *SHEIGHT* hold the width and height of the viewport in tiles, and *STX,STY* store the position of the viewport on the game screen.

One obvious snag with this idea is that the program will halt in its tracks when it reaches the end of the current map buffer. You'll now need to draw up the next section of the playing area using MAP DO, and restart the scrolling from the far right of the buffer screen. Since the map buffer is extremely large, the drawing process may take up to 20 or 30 seconds. Although you can speed things up a little with the compiler, this is still a pretty serious problem.

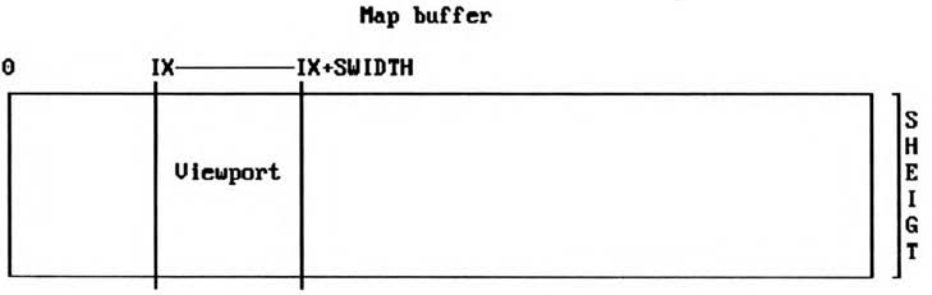

Figure 7.17: Fine scrolling

Fortunately, there's a cheat! Just design your game screens so that each complete level fits into a single buffer screen.

When the current level ends, you can now halt the program for a few moments, and load up the new level with MAP DO. While the level is being generated, the viewport can be replaced by something like:

```
<Loading Level 2>
```

Here's a detailed breakdown of the procedure.

- ❑ Define your main program screen and activate double buffering
- ❑ Create an extra large map buffer using SCREEN OPEN

```
Rem For a Horizontal scroller
Screen Open 1,800,100,8,Lowres
Screen Hide 1
```

This opens a long thin screen with dimensions 800 by 100. In order to conserve memory, it's sensible to reduce the number of colours down to just eight. This keeps the memory requirements down to a mere 36K!

```
Rem For a vertical scroller
Screen Open 1,224,800,8,Lowres
Screen Hide 1
```

This makes up a map buffer about two thirds of the total screen area, and consumes around 64K.

❑ Set the size of the viewport on the new map screen. Use something like:

```
Rem horizontal scroller
TX=0:TY=0:BX=800:BY=100: Map View TX,TY to BX,BY
TX=0:TY=0:BX=800:BY=100: Gosub TVIEW:Rem   from the simulator
Rem vertical scroller
TX=0:TY=0:BX=224:BY=800: Map View TX,TY to BX,BY
TX=0:TY=0:BX=224:BY=800: Gosub TVIEW:Rem   from the simulator
```

❑ Load your tiles and map data as normal.

❑ Draw the first section of the map with:

```
MX=0:MY=0: Map Do MX,MY
(MX=0:MY=0: Gosub MDO)
```

❑ You can now scroll your screen with:

```
Screen Copy 1,IX,IY,SW+IX,SH+IY To Logic(0),STX,STY
Screen Swap 0 : Wait Vbl
```

where *IX,IY* hold the screen coordinates of the section of the map buffer to be copied, *SW,SH* set the width and height of this area in pixels (screen points) and *STX,STY* represent the position of the viewport on your game screen.

❑ When you run out of data, you can simply pause the program and start a new level with MAP DO.

As AMOS screens are limited to a maximum of about 800 by 800 points, you'll obviously be restricted to about four game screens per level. If you've enough memory however, it's easy enough to draw the map buffer in several AMOS Basic screens simultaneously. So when you've finished the first screen, you can immediately jump to the next screen in the scrolling sequence. With the aid of this system, you can expand your levels to include 7 or even 10 screens at a time. It's sensible to arrange the contents of the map buffers so that they overlap slightly as in Figure 7.18.

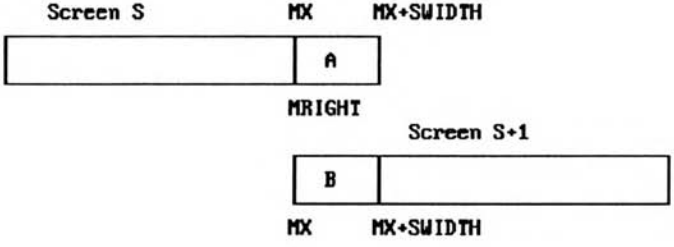

Figure 7.18: Overlapping the map buffers

Providing A and B hold identical images, you can now switch screens using a line such as:

```
If IX>=MRIGHT and CSC<2 : IX=0 : Add CSC,1,1 To 2 : End If
```

IX holds the starting point of the area to be copied from the map buffer (in screen pixels.)

CSC stores the number of the current source screen you are using for the buffer. In the current example, I'm using screens one and two for this purpose.

MRIGHT is the X coordinate of the right hand edge of the buffer area. When IX reaches this point, it's time to switch to the next buffer screen.

The *CSC<2* bit checks whether you're already on the last screen. If so, you'll need to halt your program for a few minutes and load up the next game level. Let's see how this system works out in practice.

Example 7.8: Fine scroller

```
Rem Sorry, this program requires at least a megabyte of memory
Rem It works fine on a standard A500 with an A500+1/2 meg expansion
Rem such as mine. So most users will have no problems
Rem (I'm NOT exactly a power user!)
Close Workbench : Close Editor
Double Buffer
Hide On Curs Off : Cls 0 : Centre "<Please wait>"
Rem Only use values of  16 and 32 for twidth and theight
TWIDTH=32 : THEIGHT=32
Rem This creates two massive screens!
Rem Size=800 by 352
Rem Memory=103K of CHIP memory per screen
Rem In a real game, you could limit yourself to 800*100 screens
Rem But since there's plenty of RAM, we'll have a little fun
TX=0 : TY=0 : BX=TWIDTH*25 : BY=THEIGHT*11
Rem Map buffer 1
Screen Open 1,BX,BY,8,Lowres
Screen Hide 1
Rem Map buffer 2
Screen Open 2,BX,BY,8,Lowres
Screen Hide 2
Screen 1
MAKE_ICONS[TWIDTH,THEIGHT]
Rem load map
Gosub MAP_INIT
STX=64 : STY=8 : SBX=256 : SBY=200
Gosub TVIEW
Rem Draw first half of map
MX=0 : IX=0 : IY=0 : MY=0 : Gosub TDO
Rem Draw second half
Screen 2
Rem MX ensures that the two screen overlap by 6 blocks
MX=19 : MY=0 : Gosub TDO
Rem Initialise viewport
Screen 0
Cls 0
Screen Copy 1,IX,IY,SWIDTH*TWIDTH+IX,SHEIGHT*THEIGHT+IY To
Logic(0),STX,STY
Ink 1 : Box STX-1,STY-1 To SBX+1,SBY+1
Pen 2 : Paper 0
Locate 0,0 : Pen 2 : Paper 0 : Centre "<Fine Scrolling>"
Autoback 0
```

```
Screen Swap
Rem Use for TOME
Rem Get buffer height in tiles
Rem BWIDTH=W/TWIDTH : If W mod TWIDTH<>0 Then Inc BWIDTH
Rem BHEIGHT=H/THEIGHT : If H mod THEIGHT<>0 Then Inc BHEIGHT
Rem Get height of viewport in tiles
Rem WS=Abs(SBX-STX) : HS=Abs(SBY-STY)
Rem SWIDTH=WS/TWIDTH : If WS mod TWIDTH<>0 Then Inc SWIDTH
Rem SHEIGHT=HS/THEIGHT : If HS mod THEIGHT<>0 Then Inc SHEIGHT
Rem SP=speed
Rem CSC=Current map buffer FSC=First map buffer LSC=Last...
SP=1 : CSC=1 : FSC=1 : LSC=2
Rem Get position of righthand edge of the map buffer
MRIGHT=(BWIDTH-SWIDTH)*TWIDTH
Rem Find position of bottom of the map buffer
MBOTTOM=(BHEIGHT-SHEIGHT)*THEIGHT
Rem Width of the viewport in points
SW=SWIDTH*TWIDTH
Rem Height of the viewport
SH=SHEIGHT*THEIGHT
Rem main loop
Do
   OX=IX : OY=IY
   Rem Move Up
   If Jup(1) and IY>0 Then Add IY,-SP
   Rem Down
   If Jdown(1) and IY<MBOTTOM Then Add IY,SP
   Rem Left
   If Jleft(1)
      Rem Check for an allowable movement
      If IX>0 : Add IX,-SP : End If
      Rem Flick to a new buffer screen if appropriate
      If IX<=0 and CSC>FSC : IX=MRIGHT : Add CSC,-1,FSC To LSC : End If
   End If
   Rem Right
   If Jright(1)
      Rem check movement
      If IX<MRIGHT : Add IX,SP : End If
      Rem Switch to a new buffer screen if available
      If IX>=MRIGHT and CSC<LSC : IX=0 : Add CSC,1,FSC To LSC : End If
   End If
   If OX<>IX or OY<>IY
      Rem copy map from current buffer screen to viewport
      Screen Copy CSC,IX,IY,SW+IX,SH+IY To Logic(0),STX,STY
      Screen Swap 0 : Wait Vbl
   End If
Loop
Procedure TINY_TAME[T$,W,H]
   Erase 6 : Reserve As Data 6,W*H+50
   Doke Start(6),W : Doke Start(6)+2,H
   Rem load map
   For M=0 To Len(T$)-1
      I1=Peek(Varptr(T$)+M)
      If I1=32 Then IC=0
      Rem handle numbers (All TILE values  are  ONE LESS than the icon NO
      If I1>=Asc("1") and I1<=Asc("9") Then IC=I1-Asc("1")
      Rem handle letters (A=10,B=11..etc)
      If I1>=Asc("A") and I1<=Asc("Z") Then IC=I1-Asc("A")+10
      If I1>=Asc("a") and I1<=Asc("a") Then IC=I1-Asc("a")+10
```

```
      Rem check for an error
      If IC>Length(2) or IC<0 Then IC=0
      Poke Start(6)+M+4,IC
   Next M
End Proc
Procedure MAKE_ICONS[W,H]
   Rem NEW VERSION
   Curs Off : Cls 0 : Paper 0
   Locate 0,20 : Centre "<Please wait>"
   Pen 1
   Rem first icon is a blank space
   Get Icon 1,0,0 To W,H
   Rem make some blocks
   For B=2 To 8
      Ink B : Bar 0,0 To W-1,H-1
      Ink 2 : Box 0,0 To W-1,H-1
      Get Icon B,0,0 To W-1,H-1 : Cls 0,0,0 To W+1,H+1
   Next B
End Proc
TVIEW:
Rem This routine sets up the window for TOME
S6=Start(6) : MPX=Deek(S6) : MPY=Deek(S6+2)
TWIDTH=Max(16,TWIDTH) : THEIGHT=Max(16,THEIGHT)
W=Abs(BX-TX) : H=Abs(BY-TY)
Rem Get buffer height in tiles
BWIDTH=W/TWIDTH : If W mod TWIDTH<>0 Then Inc BWIDTH
BHEIGHT=H/THEIGHT : If H mod THEIGHT<>0 Then Inc BHEIGHT
Rem Get height of viewport in tiles
WS=Abs(SBX-STX) : HS=Abs(SBY-STY)
SWIDTH=WS/TWIDTH : If WS mod TWIDTH<>0 Then Inc SWIDTH
SHEIGHT=HS/THEIGHT : If HS mod THEIGHT<>0 Then Inc SHEIGHT
Return
TDO:
Rem This routine draws the map in the buffer set by the TVIEW
Rem procedure at map co-ordinates MX,MY
S6=Start(6)+4
If MX<0 or MX>MPX or MY<0 or MY>MPY Then Return
YS=MY*MPX
Rem BHEIGHT=Height of the buffer in tiles
For Y=0 To BHEIGHT-1
   XS=MX
   Rem BWIDTH=WIDTH of the buffer
   For X=0 To BWIDTH-1
      BL=Peek(S6+(XS mod MPX)+YS)
      Paste Icon TX+X*TWIDTH,TY+Y*THEIGHT,1+BL
      Inc XS
   Next X
   Add YS,MPX
Next Y
Return
MAP_INIT:
T$=""
Rem  columns 7 To 52
Rem    01234567890123456789012345678901234567890123
T$=T$+"22222222222222222222222222222222222222222222"
T$=T$+"2                                            2" : Rem 11 rows
T$=T$+"23333333  33333333333333333333333333333333  2"
T$=T$+"2      3  4         5           6      3  2"
T$=T$+"2      3  4 44444   5 66666666  6      3  2"
T$=T$+"2         4 4   4   5        6  6      3  2"
```

```
T$=T$+"2       3  4 444 4   5         6   6       3  2"
T$=T$+"2       3  4     4   5 66666666   6       3  2"
T$=T$+"23333333  4444444   5              6       3  2"
T$=T$+"2                                                2"
T$=T$+"22222222222222222222222222222222222222222222222"
TINY_TAME[T$,44,11]
Return
```

As before, I'll now list the advantages and the disadvantages of this system.

Advantages

- Scrolling is ultra smooth and the speed can be changed at any time. So it's perfect for a pure vertical or horizontal scroller.

Disadvantages

- The finished program will ONLY work on 1 meg machines.
- If the scrolling is in several directions, the size of the buffer can expand to ridiculous proportions.

Warning! The reliability of the AMOS icon routines cannot be guaranteed for REALLY large screens. If it works once, you're probably safe, but ALWAYS save your current program before experimenting with a new screen setting. It's also advisable to reset the screen using DEFAULT before returning to the editor. Don't ask me why, but it seems to keep the dreaded GURU message away nicely.

Two phase scrolling

As its name suggests, this is a simple cross between the previous two scrolling systems. The hidden map buffer is expanded to be one tile larger than the visible screen (viewport), as shown in Figure 7.19.

W and *H* represent the width and height of a single map tile. The viewport can then be gently scrolled through the map buffer until it reaches one of the edges. This can be accomplished with a SCREEN COPY command such as:

```
Screen Copy 1,STX+FX,STY+FY,SBX+FX,SBY+FY To Logic(0),STX,STY
```

FX and *FY* now hold the distance between the viewport and the edge of the map window. When the viewport reaches a screen boundary, the map buffer is quickly scrolled to the next section of the map using a coarse scroller like the one in Example 7.7. The viewport is then repositioned at the top left corner of this screen, and the process is repeated. It's important to recognise that the scrolling is performed in two separate phases.

- The fine scroller moves the viewport through the current section of the playing area. This is taken straight from the existing image held in the map buffer.
- The coarse scroller however, moves the invisible map buffer through the playfield, a tile at a time.

So much for the theory. On the next page, there's an example.

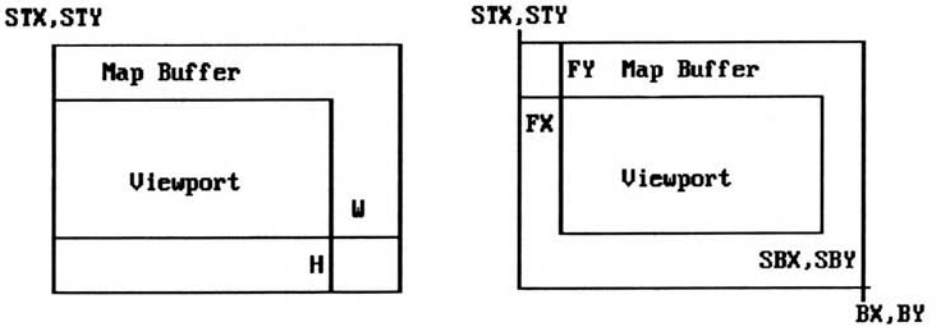

Figure 7.19: Left: Two phase scrolling; right: scrolling zone

Example 7.9: Two phase scroller

```
Double Buffer
Curs Off : Cls 0 : Centre "<Please wait>"
Rem Only use values of  16 and 32 for twidth and theight
TWIDTH=32 : THEIGHT=32
Screen Open 1,320,200,16,Lowres
Screen Hide 1
MAKE_ICONS[TWIDTH,THEIGHT]
Gosub MAP_INIT
STX=80 : STY=32 : SBX=240 : SBY=160
TX=STX : TY=STY : BX=SBX+TWIDTH : BY=SBY+THEIGHT
Gosub TVIEW
Cls 0
MX=10 : IX=10*TWIDTH : MY=0 : Gosub TDO
Screen Copy 1,STX,STY,SBX,SBY To Logic(0),STX,STY
Screen 0
Ink 1 : Box STX-1,STY-1 To SBX+1,SBY+1
Pen 2 : Paper 0
Locate 0,0 : Pen 2 : Paper 0 : Centre "<Two phase scrolling>"
Autoback 0
Screen Swap
Screen Copy Physic(0) To Logic(0)
SP=1 : Rem speed
Rem Main loop
Do
   Rem IX,IY are the coordinates of the section of the map we are
   Rem currently viewing
   OX=IX : OY=IY
   If Jleft(1) and IX>0
      Add IX,-SP
      Rem MX holds the X coordinate of the area held in the map buffer
      Rem It's converted into a MAP coordinate (in units of a tile)
      MX=IX/TWIDTH
      Rem Have we reached the left hand edge of the map buffer?
      Rem This gets remainder when IX is divided by TWIDTH
      Rem and scrolls when it's equal to TWIDTH-SP
      Rem I'm using TWIDTH-SP rather than TWIDTH
      Rem because I've just subtracted SP from IX
      If IX mod TWIDTH=TWIDTH-SP
```

```
      Rem if we're over the edge, coarse scroll the map buffer left,
      Rem and reveal the next section of the map
      Screen Copy 1,TX,TY,BX-TWIDTH,BY To 1,TX+TWIDTH,TY
      Screen 1 : Gosub TLEFT
      Rem Map Left MX,MY
      End If
   End If
   If Jright(1) and MX<MPX-SWIDTH+1
      Add IX,SP
      MX=IX/TWIDTH
      Rem have we reached the righthand edge of the map buffer
      If IX mod TWIDTH=0
      Rem Coarse Scroll the map right
      Screen Copy 1,TX+TWIDTH,TY,BX,BY To 1,TX,TY
      Screen 1 : Gosub TRIGHT
      Rem Map Right MX,MY
      End If
   End If
   If Jup(1) and IY>0
      Add IY,-SP
      Rem MY holds the Y coordinate of the current section of the map
      Rem measured in units of a single tile
      MY=IY/THEIGHT
      Rem Have we exceeded the TOP of the buffer?
      Rem Gets remainder of IY/THEIGHT and jumps when it's THEIGHT-SP
      If IY mod THEIGHT=THEIGHT-SP
      Rem Coarse scroll the map buffer up
      Screen Copy 1,TX,TY,BX,BY-THEIGHT To 1,TX,TY+THEIGHT
      Screen 1 : Gosub TTOP
      Rem Map Top MX,MY
      End If
   End If
   If Jdown(1) and MY<MPY-SHEIGHT+1
      Add IY,SP
      MY=IY/THEIGHT
      Rem have we reached the bottom of the map buffer?
      If IY mod THEIGHT=0
      Rem Coarse scroll the map down
      Screen Copy 1,TX,TY+THEIGHT,BX,BY To 1,TX,TY
      Screen 1 : Gosub TBOTTOM
      Rem Map Bottom MX,MY
      End If
  End If
 Rem Fine scroller!
 Rem Check for a movement
  If OX<>IX or OY<>IY
      Rem Get distance from the edge of the map buffer
      FX=IX mod TWIDTH : FY=IY mod THEIGHT
      Rem copy map buffer into viewport
      Screen Copy 1,STX+FX,STY+FY,SBX+FX,SBY+FY To Logic(0),STX,STY
      Rem Switch logical and physical screens
      Screen Swap 0 : Wait Vbl
   End If
Loop
Procedure MAKE_ICONS[W,H]
   Rem NEW VERSION!
   Curs Off : Cls 0 : Paper 0
   Locate 0,20 : Centre "<Please wait>"
   Pen 15
   Get Icon 1,0,0 To W,H
```

```
   Rem make some blocks
   For B=2 To 8
      Ink B+4 : Bar 0,0 To W-1,H-1
      Ink 2 : Box 0,0 To W-1,H-1
      Get Icon B,0,0 To W-1,H-1
      Cls 0,0,0 To W+1,H+1
   Next B
End Proc
Procedure TINY_TAME[T$,W,H]
   Erase 6 : Reserve As Data 6,W*H+50
   Doke Start(6),W : Doke Start(6)+2,H
   Rem load map
   For M=0 To Len(T$)-1
      I1=Peek(Varptr(T$)+M)
      If I1=32 Then IC=0
      Rem handle numbers (All TILE values  are  ONE LESS than the ico
      If I1>=Asc("1") and I1<=Asc("9") Then IC=I1-Asc("1")
      Rem handle letters (A=10,B=11..etc)
      If I1>=Asc("A") and I1<=Asc("Z") Then IC=I1-Asc("A")+10
      If I1>=Asc("a") and I1<=Asc("a") Then IC=I1-Asc("a")+10
      Rem check for an error
      If IC>Length(2) or IC<0 Then IC=0
      Poke Start(6)+M+4,IC
   Next M
End Proc
TVIEW:
Rem This routine sets up the window for TOME
S6=Start(6) : MPX=Deek(S6) : MPY=Deek(S6+2)
TWIDTH=Max(16,TWIDTH) : THEIGHT=Max(16,THEIGHT)
W=Abs(BX-TX) : H=Abs(BY-TY)
Rem Get buffer height in tiles
SWIDTH=W/TWIDTH : If W mod TWIDTH<>0 Then Inc SWIDTH
SHEIGHT=H/THEIGHT : If H mod THEIGHT<>0 Then Inc SHEIGHT
Return
TDO:
Rem This routine draws the map in the buffer set by the TVIEW
Rem procedure at map co-ordinates MX,MY
S6=Start(6)+4
If MX<0 or MX>MPX or MY<0 or MY>MPY Then Return
YS=MY*MPX
Rem SHEIGHT=Height of the buffer in tiles
For Y=0 To SHEIGHT-1
   XS=MX
   Rem SWIDTH=WIDTH of the buffer
   For X=0 To SWIDTH-1
      BL=Peek(S6+(XS mod MPX)+YS)
      Paste Icon TX+X*TWIDTH,TY+Y*THEIGHT,1+BL
      Inc XS
   Next X
   Add YS,MPX
Next Y
Return
TLEFT:
S6=Start(6)+4
YS=MY*MPX
XS=MX
For Y=0 To SHEIGHT-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon TX,TY+Y*THEIGHT,1+BL
   Add YS,MPX
```

```
Next Y
Return
TRIGHT:
Rem Map Right
S6=Start(6)+4
YS=MY*MPX
XS=MX+SWIDTH-1
For Y=0 To SHEIGHT-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon BX-TWIDTH,TY+Y*THEIGHT,1+BL
   Add YS,MPX
Next Y
Return
TTOP:
Rem Map Top
S6=Start(6)+4
YS=MY*MPX
XS=MX
For X=0 To SWIDTH-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon TX+X*TWIDTH,TY,1+BL
   Inc XS
Next X
Return
TBOTTOM:
Rem Map Bottom
S6=Start(6)+4
YS=(MY+SHEIGHT-1)*MPX
XS=MX
For X=0 To SWIDTH-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon TX+X*TWIDTH,BY-THEIGHT,1+BL
   Inc XS
Next X
Return
MAP_INIT:
T$=""
Rem    01234567890123456789012345678901234567890123456789
T$=T$+"22222222222222222222222222222222222222222222222" : Rem  columns
T$=T$+"2                                                2" : Rem 11 rows
T$=T$+"23333333  33333333333333333333333333333333  2"
T$=T$+"2       3  4          5             6       3  2"
T$=T$+"2       3  4 44444    5 66666666    6       3  2"
T$=T$+"2          4 4   4    5          6  6       3  2"
T$=T$+"2       3  4 444 4    5          6  6       3  2"
T$=T$+"2       3  4     4    5 66666666    6       3  2"
T$=T$+"23333333  4444444    5             6       3  2"
T$=T$+"2                                                2"
T$=T$+"22222222222222222222222222222222222222222222222"
TINY_TAME[T$,44,11]
Return
```

A version of this system is used by the programming examples on the AMOS TOME disk. The programming may look very different, but the principles are identical.

Advantages

- Fast,and reasonably efficient, this routine gives best results when the programs scrolling by a single tile. So it's fine for RPGs or Platform games.

Disadvantages

- There's a slight jerk to the movement when the coarse scroller fires up to display the next tile in the map. This can be reduced by increasing the sizes of your map tiles, but it's hard to get rid of completely, even with the compiler.

Logic/Physic scrolling

The final example is effectively a compromise between the three routines I've shown you up to now. It uses an extended version of the TOME simulation routines to perform all our scrolling on a single screen. This allows us to generate smooth scrolling effects while using relatively modest amounts of memory.

Before I go into the details, I'll first introduce you to the actual scrolling routines. Although the theory behind them is fairly advanced, they are quite easy to use in a real program. So feel free to call these routines in your own games, even if the reasoning eludes you.

Horizontal scrolling

SLEFT

SLEFT scrolls the viewport one place to the left. The movement speed is measured in single screen points, and is read from the variable SP. The coordinates of the current section of the map to be displayed in the viewport are held in *IX* and *MY*

IX is the horizontal distance from the beginning of the map to the start of the visible playing area. Like the speed, it's measured in pixels.

MY is a map coordinate which stores the position of the viewport from the top of the map. In a horizontal scrolling game, *MY* can be safely set to zero. It's only needed if the playfield is taller than the visible viewing area.

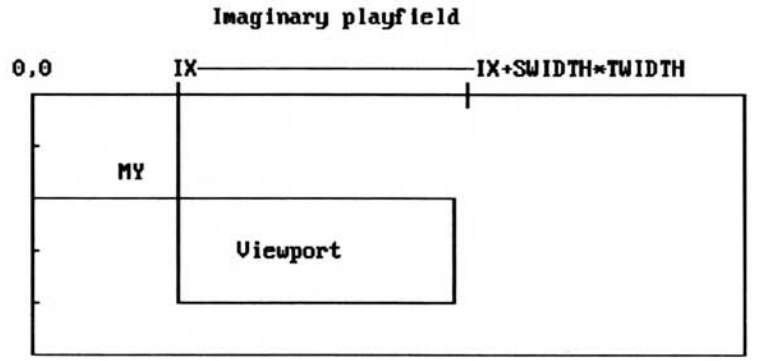

Figure 7.20: Logic/physic scrolling - 1

I've written SLEFT as a standard AMOS subroutine. So once you've loaded it into the editor, it can be called straight from AMOS Basic using a GOSUB command like:

```
SP=1 : MY=0 : IX=10 : Gosub SLEFT
```

There's also similar routine to scroll the map to the right.

SRIGHT
Moves the viewport one place to the right of the playfield. Aside from the direction though, SRIGHT is identical to its left handed cousin.

```
SP=1 : MY=0 : IX=9 : Gosub SRIGHT
```

Vertical scrolling

SUP
Scrolls the viewport up by a single screen point.

SDOWN
Moves the viewport downwards.

SUP and SDOWN use the following variables:

SP controls the scrolling speed. The higher the value, the more points which will be moved in each operation.

MX holds the map coordinate of the left edge of the playfield, and *IY* stores the screen coordinate of the vertical segment of the map to be displayed. If you are only scrolling vertically, *MX* will always be zero. It's included solely to allow SUP and SDOWN to be used with either SLEFT or SRIGHT. This enables us to combine vertical and horizontal scrolling routines in a single game.

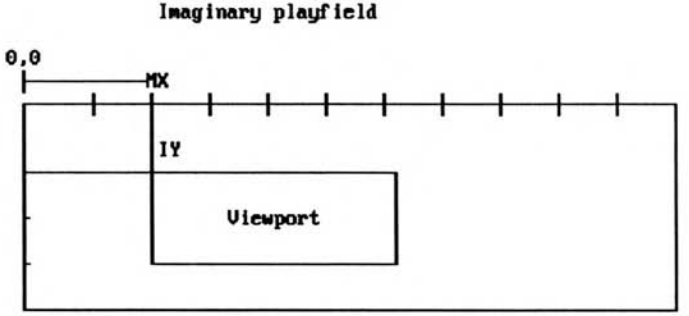

Figure 7.21: Logic/physic scrolling - 2

I'll now go through the techniques required to use these routines in one of your games. The process is quite involved, but it can be accomplished in less than fifteen lines. So once you've got the hang of it, you should find it reasonably easy.

Using a logic/physic scroller

❑ Start off by creating a screen to hold your viewport as in Figure 7.22. This should be contain an invisible buffer zone of one tile in each of the directions you will be scrolling.

Use a command like:

```
Screen Open 0,320+TWIDTH*2,256+THEIGHT*2,8
```

❑ Now centre the viewing area on the expanded screen. This involves two separate instructions.

SCREEN DISPLAY sets the size and position of the current screen on your TV. The format is:

SCREEN DISPLAY screen,X,Y,WIDTH,HEIGHT

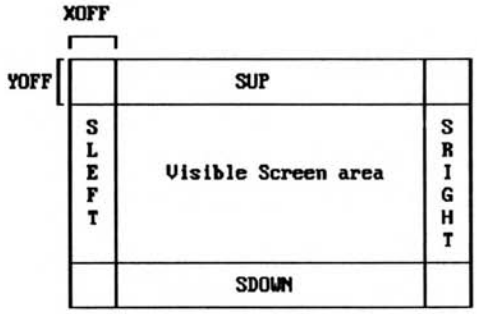

Figure 7.22: Setting an invisible buffer area

The normal screen position is at (128,50),so we can set the size with:

```
Screen Display 0,128,50,320,256
```

If we are intending to scroll the map left or up, we'll now need to move the display to the starting point of the visible screen area with SCREEN OFFSET. This will ensure that the regions at the top left of the viewport will be totally hidden from view.

```
SCREEN OFFSET screen,xoff,yoff
```

Screen offset chooses the section of a screen which will be displayed. *xoff* and *yoff* hold the starting coordinates of the section of the image which will be displayed. Any areas below these coordinates will disappear completely off the edge of the screen.

SCREEN OFFSET is quite a powerful AMOS command, and it can actually be used to generate its own impressive range of scrolling effects. I'll be giving it the attention it deserves in Section 7.5.

In our current example however, we can get away with either:

```
Screen Offset 0,TWIDTH,0:Rem for SLEFT
Screen Offset 0,0,THEIGHT:Rem for SUP
Screen Offset 0,TWIDTH,THEIGHT: Rem for SLEFT AND SUP
```

- ❑ Activate double buffering with DOUBLE BUFFER
- ❑ Turn off the automatic background system with AUTOBACK 0
- ❑ Define the rectangular area on the screen where the map will be physically drawn. Use:

```
TX=0 : TY=0 : BX=352+32 : BY=288+32
STX=TX+SP : STY=TY+SP : SBX=BX : SBY=BY
Gosub TVIEW:Rem or Map View TX,TY To BX,BY
```

- ❑ Draw the first section of the map with MAP DO or TDO:

```
MX=0 : MY=0 : IY=MY*THEIGHT : IX=0 : Gosub TDO
Rem Map Do MX,MY
```

- ❑ Copy the new viewport into both the physical and logical screens. Use something like:

```
Screen Swap : Wait Vbl
Screen Copy Physic To Logic
```

❑ Set up the scrolling speed in *SP*, and scroll the map in the required direction. This will prime the logical and physical screens for use by the main scrolling routine.

```
SP=1
Gosub SDOWN
Screen Swap : Wait Vbl
```

❑ Finally, multiply *SP* by 2 and repeatedly scroll your screen in the appropriate direction.

```
IP=SP
SP=IP*2
IY=IY+SP
```

Whew! Well, that's the Basic recipe. Here's one I made earlier!

Example 7.10: Logic/physic scrolling through a map

```
Rem Example 7.10 Logic/Physic scrolling
Rem Open the screen
Screen Open 0,320+32*2,192+32*2,8,Lowres
Hide On
Double Buffer
Rem Set up screen display
Screen Display 0,128,60,320,192
Screen Offset 0,32,32
Rem Only use values of  16 and 32 for twidth and theight
TWIDTH=32 : THEIGHT=32
Rem Create MAP TILES
Curs Off : Cls 0
Screen Open 1,320,10,8,Lowres
Screen Display 1,128,50,320,10
Curs Off : Cls 0 : Paper 0 : Centre "<Please wait>"
Screen Hide 0
Screen 0
Rem Set speed. Use speeds like 1,2,4,8 or 16
SP=2
Rem NEW version
MAKE_ICONS[TWIDTH,THEIGHT]
Rem Create map
Gosub MAP_INIT
Rem Set up viewing area
TX=0 : TY=0 : BX=352+32 : BY=192+64
STX=TX+SP : STY=TY+SP : SBX=BX : SBY=BY
Gosub TVIEW
Rem Map View TX,TY To BX,BY
Curs Off : Flash Off : Cls 0
Rem Turn off autoback
Autoback 0
Rem Draw first bit of the map
MX=0 : MY=0 : IY=MY*THEIGHT : IX=0 : Gosub TDO
Rem Map Do MX,MY
Rem Set up border lines or playfield
SW=SWIDTH*TWIDTH
SH=SHEIGHT*THEIGHT
MTOP=IY
MRIGHT=(MPX-SWIDTH-1)*TWIDTH
MBOTTOM=(MPY-SHEIGHT)*THEIGHT
```

```
MTOP=IY+THEIGHT
MLEFT=IX+TWIDTH
Screen 1 : Centre "<Logic/Physic scrolling>" : Screen 0 : Screen Show 0
Rem Copy first section of MAP into both physical and logical screens
Screen Swap : Wait Vbl
Screen Copy Physic To Logic
Rem Save old speed in IP
IP=SP
Rem Scroll speed one place down
Gosub SDOWN
Screen Swap : Wait Vbl
Rem Double scrolling steps
SP=IP*2
IY=IY
Rem Scroll screen down
Rem I'm using Repeat...Until rather than For..Next due
Rem to a minor bug in the AMOS compiler.
Rem It doesn't allow you to call GOSUB inside a FOR..NEXT. Whoops!
Repeat
   Gosub SDOWN
   Screen Swap : Wait Vbl
   Rem Move Viewport one place downwards
   Add IY,IP
Until IY=MBOTTOM-IP
Bell
Rem change direction!
Rem Reset screen
Screen Copy Physic To Logic
Rem Reset coordinates
MY=MY+1
Rem Prime logical and physical screens for SRIGHT
SP=IP : Rem Original speed was in IP
Gosub SRIGHT
Screen Swap
Wait Vbl
Rem Double speed for logic/physic scrolling
SP=IP*2
SX=IX
Rem Scroll right!
Repeat
   Gosub SRIGHT
   Screen Swap : Wait Vbl
   Rem Move Viewport one place to the right
   Add IX,IP
Until IX=MRIGHT-IP
Bell
Rem Change direction
Rem prime screens for SUP
Screen Copy Physic To Logic
SP=IP
MX=MX+1
IY=MBOTTOM-THEIGHT*2
Rem Scroll logic screen one place up
Gosub SUP
Screen Swap : Wait Vbl
Rem double scrolling size
SP=IP*2
Rem scroll up!
Repeat
   Gosub SUP
```

```
   Screen Swap : Wait Vbl
   Rem Move viewport up
   Add IY,-IP
Until IY=MTOP+IP
Rem Prime screens for SLEFT
Screen Copy Physic To Logic
Bell
SP=IP : Rem get old speed
Dec MY
Rem Scroll IP units left
IX=MRIGHT+TWIDTH
Gosub SLEFT
Screen Swap
Wait Vbl
Rem Double speed
SP=IP*2
Rem Scroll LEFT
Repeat
   Gosub SLEFT
   Screen Swap : Wait Vbl
   Rem Move viewport IP units left
   Add IX,-IP
Until IX=MLEFT+IP
Bell
Stop
Rem Da simulator code!
SUP:
MY=IY/THEIGHT : Rem Convert IY into a tile coordinates
Rem get number of points in the current tile to display.
FY=(IY mod THEIGHT)
Rem Get offsets to start of map area
XS=MX mod MPX
YS=(MY-1)*MPX
Rem Scroll screen with SCREEN COPY
Screen Copy Logic(0),TX,TY,BX,BY To Logic(0),TX,TY+SP
S6=Start(6)+4
Rem calculate position of next line of tiles to draw
PY=TY+SP-FY
Rem Draw tiles along the top
For F=0 To SWIDTH-1
   B=Peek(S6+XS+YS+F mod MPX)
   Paste Icon TX+F*TWIDTH,PY,B+1
Next F
Return
SDOWN:
Rem scroll the screen down with screen copy
Screen Copy Logic(0),TX,TY+SP,BX,BY To Logic(0),TX,TY
S6=Start(6)+4
Rem convert IY into a map coord
MY=IY/THEIGHT
Rem Get number of points in the lower tiles to display
FY=IY mod THEIGHT
Rem find start of tiles in memory
XS=MX mod MPX
YS=(MY+SHEIGHT)*MPX
Rem position tiles along the bottom of the screen
PY=BY-SP-FY
Rem Draw bottom row of tiles
For F=0 To SWIDTH-1
   B=Peek(S6+XS+YS+F mod MPX)
```

```
   Paste Icon TX+F*TWIDTH,PY,B+1
Next F
Return
SLEFT:
Rem scroll screen
Screen Copy Logic(0),TX,TY,BX,BY To Logic(0),TX+SP,TY
S6=Start(6)+4
Rem Convert IX into a map coordinate
MX=IX/TWIDTH
Rem find area of the tile to the left of the screen
FX=IX mod TWIDTH
Rem First start of the current tiles
YS=MY*MPX
XS=MX-1
Rem position tiles
PX=TX-FX+SP
Rem draw tiles
For Y=0 To SHEIGHT-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon PX,TY+Y*THEIGHT,1+BL
   Add YS,MPX
Next Y
Return
SRIGHT:
Rem Copy the screen one place to the left
Rem If this sounds crazy, remember that the background moves
Rem in the opposite direction to the apparent motion. So we really are
Rem scrolling RIGHT,right
Screen Copy Logic(0),TX+SP,TY,BX,BY To Logic(0),TX,TY
Rem Initialise MAP reader
S6=Start(6)+4
Rem this represents the number of points of the tiles which
Rem will be drawn from the left
FX=IX mod TWIDTH
Rem convert IX into a map coordinate
MX=IX/TWIDTH
Rem Find the line of map tiles in the memory bank
YS=MY*MPX
XS=MX+SWIDTH
Rem set drawing position of the tiles
PX=BX-FX-SP
Rem Draw the tiles
For Y=0 To SHEIGHT-1
   BL=Peek(S6+(XS mod MPX)+YS)
   Paste Icon PX,TY+Y*THEIGHT,1+BL
   Add YS,MPX
Next Y
Return
Procedure TINY_TAME[T$,W,H]
   Erase 6 : Reserve As Data 6,W*H+50
   Doke Start(6),W : Doke Start(6)+2,H
   For M=0 To Len(T$)-1
      I1=Peek(Varptr(T$)+M)
      If I1=32 Then IC=0
      If I1>=Asc("1") and I1<=Asc("9") Then IC=I1-Asc("1")
      If I1>=Asc("A") and I1<=Asc("Z") Then IC=I1-Asc("A")+10
      If I1>=Asc("a") and I1<=Asc("a") Then IC=I1-Asc("a")+10
      If IC>Length(2) or IC<0 Then IC=0
      Poke Start(6)+M+4,IC
   Next M
```

```
End Proc
Procedure MAKE_ICONS[W,H]
   Curs Off : Flash Off : Cls 0 : Paper 0
   Pen 1
   Rem first icon is a blank space
   Get Icon 1,0,0 To W,H
   Rem make some balls!
   For B=2 To 8
      Ink 1 : Circle W/2,H/2,H/2
      Ink B : Paint W/2,H/2
      Get Icon B,0,0 To W,H : Cls 0,0,0 To W+1,H+1
   Next B
End Proc
TVIEW:
Rem This routine sets up the size and position of the map buffer
S6=Start(6) : MPX=Deek(S6) : MPY=Deek(S6+2)
TWIDTH=Max(16,TWIDTH) : THEIGHT=Max(16,THEIGHT)
W=Abs(BX-TX) : H=Abs(BY-TY)
VWIDTH=W/TWIDTH : If W mod TWIDTH<>0 Then Inc VWIDTH
VHEIGHT=H/THEIGHT : If H mod THEIGHT<>0 Then Inc VHEIGHT
WS=Abs(SBX-STX) : HS=Abs(SBY-STY)
SWIDTH=WS/TWIDTH : If WS mod TWIDTH<>0 Then Inc SWIDTH
SHEIGHT=HS/THEIGHT : If HS mod THEIGHT<>0 Then Inc SHEIGHT
Return
TDO:
Rem This routine draws the map in the window set by the TVIEW
Rem Procedure
Rem at map co-ordinates MX,MY
S6=Start(6)+4
If MX<0 or MX>MPX or MY<0 or MY>MPY Then Return
YS=MY*MPX
For Y=0 To SHEIGHT-1
   XS=MX
   For X=0 To SWIDTH-1
      BL=Peek(S6+(XS mod MPX)+YS)
      Paste Icon TX+X*TWIDTH,TY+Y*THEIGHT,1+BL
      Inc XS
   Next X
   Add YS,MPX
Next Y
Return
MAP_INIT:
T$=""
Rem    01234567890123456789012345678901234567890123456789012345678901234567890123456789
T$=T$+"22222222222222222222222222222222222222222222222222222222222222222222222222222"
T$=T$+"2                                                                             2"
T$=T$+"24444445   44444444444444444444444444444444444444444444444444444444444444  2"
T$=T$+"2      5   4         5      6                                              5  2"
T$=T$+"2      5   4 44444   5      666666666666666666666666666                  5  2"
T$=T$+"2          4 4   4   5             6        6        6                   5  2"
T$=T$+"2      5   4 444 4   5      66666666   6    6    6   6                   5  2"
T$=T$+"2      5   4     4   5                 6         6   6                   5  2"
T$=T$+"24444445   4444444   5      666666666666666666666666666                  5  2"
T$=T$+"2                                                                        5  2"
T$=T$+"2    44444444444444444444444444444444444444444444444444444444444444444  2"
T$=T$+"2                                                                             2"
T$=T$+"2                                                                             2"
T$=T$+"2                                                                             2"
T$=T$+"24444445   44444444444444444444444444444444444444444444444444444444444444  2"
T$=T$+"2      5   4         5      6                                              5  2"
```

```
T$=T$+"2      5   4 44444   5      6666666666666666666666666  5  2"
T$=T$+"2          4 4   4   5            6       6        6  5  2"
T$=T$+"2      5   4 444 4   5      66666666   6   6    6   6  5  2"
T$=T$+"2      5   4     4   5                 6       6   6  5  2"
T$=T$+"24444445   4444444   5      6666666666666666666666666  5  2"
T$=T$+"2                                                       5  2"
T$=T$+"2    444444444444444444444444444444444444444444444444  2"
T$=T$+"2    5                                                  2"
T$=T$+"22222222222222222222222222222222222222222222222222222222"
TINY_TAME[T$,60,25]
Return
```

The theory behind this routine is complex, but it's very similar to the message scroller in Example 7.5. Each subroutine scrolls the screen in the required direction using SCREEN COPY, and then draws in the new line of tiles along the appropriate edge of the screen. The position of these tiles is loaded into either PX or PY depending on the direction.

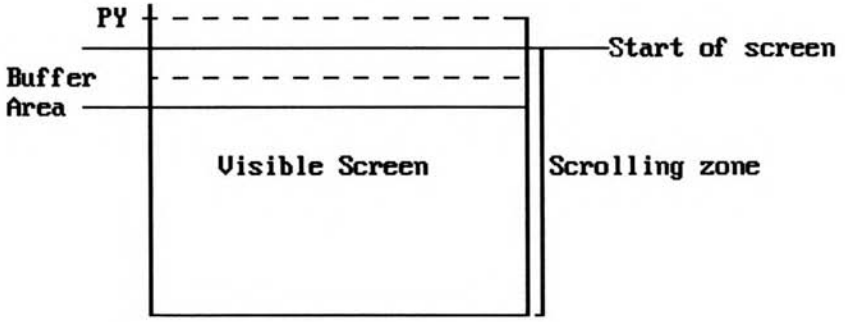

Figure 7.23: SUP in action

Let's take the case of SUP - Figure 7.23. This first scrolls the contents of the scrolling zone downwards, and then draws the line of tiles at *PY*.

```
PY=TY+SP-FY
```

The tiles are drawn starting at *PY,0*. *PY* usually refers to a point which is ABOVE the screen, and it's therefore negative! AMOS Basic will automatically ignore any parts of our graphics which lie outside the current screen boundaries. So by loading PY with a negative value, we can ensure that only a slice of our tiles will be displayed in the buffer zone.

Note that since all the scrolling operations are performed off-screen, I've managed to get rid of the separate SETUP routine I used in Example 7.5 (BANNER_LSET). By changing the size of the scrolling area slightly, it's also been possible to use the same value (*SP*) for both operations. This streamlines the entire scrolling system, and reduces the size of our program considerably.

MX or *MY* hold the map coordinates of the line of tiles which are currently being drawn along the appropriate screen boundary. These are used in conjunction with the *XS* and *YS* variables to determine the position of the map data in memory. MPX holds the width of the map in tiles.

```
XS=MX mod MPX
YS=(MY-1)*MPX
```

Whenever the map passes over a complete tile, *MX* or *MY* will change, and the scrolling routine will move directly to the next line of tiles in the map.

By the way, don't get confused with the scrolling directions. In order to scroll the viewport upwards through the playfield, we need to move the existing section downwards. This makes room for the new sections of the map which appear from the top. OK?

7.5 Screen Offset Scrolling (Hardware Scrolling)

Now for something COMPLETELY different! As we've seen, AMOS is capable of generating massive screens which would be impossible to display on your TV in a single go.

If AMOS encounters one of these extra large screens, it will automatically position the display over top left hand corner of the image as in Figure 7.24. So we only see a fraction of the total screen area at any one time.

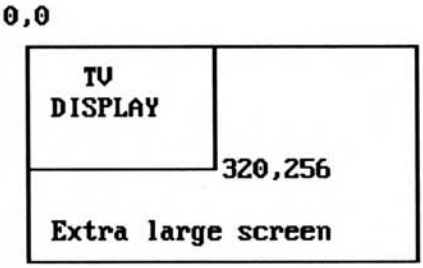

Figure 7.24: The display of an extra large screen

AMOS Basic provides a simple SCREEN OFFSET command which allows us to move this display window over the entire screen. It works using a system of offset values like so:

SCREEN OFFSET screen,xoff,yoff

SCREEN OFFSET moves the starting point of the display from coordinates 0,0 to a new screen position given by the values *xoff* and *yoff*.

After calling this instruction, the coordinates of the top left hand corner of the display area will change from 0,0 to *xoff,yoff* as in Figure 7.25. So if for example, you've positioned a Blitter object at 0,0, it will immediately disappear.

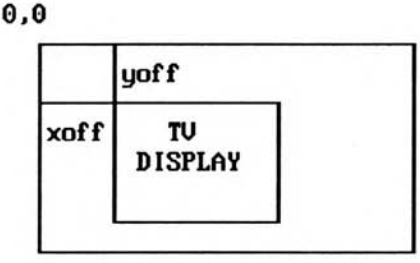

Figure 7.25: The display moves to coordinates xoff,yoff

Note that there's absolutely NO effect on the actual contents of the screen. Only the display position is changed. Once we've drawn our map, we can scroll it through the display using a simple loop such as:

```
For X=0 to W
  Screen Offset 1,X,0:Wait Vbl
Next X
```

The WAIT VBL command incidentally is needed to slow the scrolling down so we can actually SEE it on the screen!

Let's look at a small example of this command. We'll take the screen scroller from Example 7.1, and modify it for use with hardware scrolling.

Example 7.11: Hardware scrolling

```
Rem Reserve some space
Close Editor
Close Workbench
Curs Off : Cls 0 : Paper 0 : Hide
Rem create the extra large screen
Screen Open 1,960,50,8,Lowres
Rem only display a 320 wide section at a time.
Rem This is important, AMOS normally tries to cram as much
Rem as it can on the screen at once. The best results are obtained
Rem if the display is a maximum of 320 points wide and 256 high
Rem I'm also taking the opportunity to move the screen to the middle
Rem of the TV display. That's at hardware coordinates 128,150
Screen Display 1,128,150,320,50
Curs Off : Cls 0 : Paper 0
Rem create a banner on the screen
BANNER["  AMOS BASIC  AMOS BASIC "]
Rem print the tile line on screen 0
Screen 0
Locate 0,0 : Pen 3 : Centre "Screen Offset Scrolling"
Rem set up an attractive rainbow effect
Colour 2,0 : Set Rainbow 1,2,32,"(2,1,16)","(2,1,16)","(2,1,16)"
Rainbow 1,0,150,32
Rem scroll the screen using SCREEN OFFSET
Do
   For SPEED=1 To 16
      For CS=0 To 12*40-1 Step SPEED
         Rem Move the display window to the section of the screen
         Rem starting at coordinates CS,0
         Screen Offset 1,CS,0
         Wait Vbl:Rem Remove this to see the REAL speed!
      Next CS
   Next SPEED
Loop
Procedure BANNER[A$]
   Rem create a simple banner
   Screen 0: Print A$
   Zoom 0,0,0,(Len(A$)-1)*8,8 To 1,0,0,959,32
   Screen 1
End Proc
```

The scrolling operation was performed by the command:

```
Screen Offset 1,CS,0
```

CS holds the current offset value. As it increases, the message is gently moved through the display area. The same idea could also be used in a real game.

But wait a minute. If hardware scrolling is so easy, why we have bothered with all this complicated Blitter scrolling? Well, although SCREEN OFFSET is VERY powerful, it does have a number of severe limitations. These rule it completely out of contention for many types of game.

Disadvantages of hardware scrolling

- ❑ SCREEN OFFSET is only capable of scrolling a complete AMOS screen. So there's no way to scroll just a section of an existing screen.
- ❑ SCREEN OFFSET doesn't work particularly well with either Blitter Objects or sprites.

Sprites are generated with the same hardware as the offset scrolling effects. One unfortunate side effect of this, is that Amiga is not able to display sprites 6 or 7 on a scrolling background. It's possible to avoid this problem by moving hardware sprites 6 and 7 off the edge of the screen using a line like:

```
Sprite 6,-100,-100,1: Rem where image 1 is a 32x32 image
```

You can use the computed sprites in the normal way. You can also see why I've avoided sprites in many of my previous programming examples.. Technical isn't it?

Bobs work fine, except that they require you to DOUBLE BUFFER the entire scrolling area. This doubles the amount of memory used, and consumes enormous quantities of RAM. In Example 7.8 for instance, I used SCREEN COPY to scroll through two 800x352 screens. These took 103K per screen with the Blitter system, but they'd require 206K each if we were scrolling them with SCREEN OFFSET.

- ❑ Another problem, is that the coordinates of the visible region of the screen are changing continually. This means that any Blitter objects will move automatically in the current scrolling direction.

Sometimes this is ideal, but usually it adds a big complication to your programs movement routines. Supposing you were writing a horizontal scrolling arcade game. Generally, these games position the player's ship somewhere near the centre of the playing area, while the background moves smoothly underneath. With SCREEN OFFSET, the ship would behave if it were a fixed part of the scrolling area. So you'd now need to move the ship slowly forwards to compensate with the scrolling effect. The same problem would also apply to all your missiles! Ouch!

This is particularly annoying if you're upgrading a static arcade game to a screen scroller, as it forces you to make massive changes to all your existing attack waves. In contrast, Blitter scrolling copies the map onto an essentially fixed viewport, so it's capable of using these waves with only minor modifications. This makes it very easy to expand a static game into a full blown screen scroller.

7.6 Parallax scrolling

Up until now, we've assumed that the background lies directly alongside the player's ship. This allows us to treat it as a single continuous landscape. In reality however, the different parts of the scene would appear to move away at slightly different speeds, depending their relative distances from the player.

You'll probably have observed this parallax effect during a real journey. The further an object lies from your viewpoint, the slower it apparently seems to move. Obviously, the parallax effect is just a trick of the light, as the background isn't actually moving anywhere. It's sitting there, solid as a rock, while you move off in front of it. The best commercial arcade games incorporate this parallax into their scrolling routines, adding a considerable amount of extra realism to their graphics. I'll now show you a couple of techniques which can be used to generate these effects in your own AMOS Basic programs.

7.6.1 Multiple screens

If your landscape is fairly flat, you may be able to split up your scrolling area into several discrete layers as shown in Figure 7.26. Each layer represents the section of the horizon at a particular distance from the player. Providing the layers don't overlap, you can now simply scroll each section at a different speed, creating an attractive parallax effect.

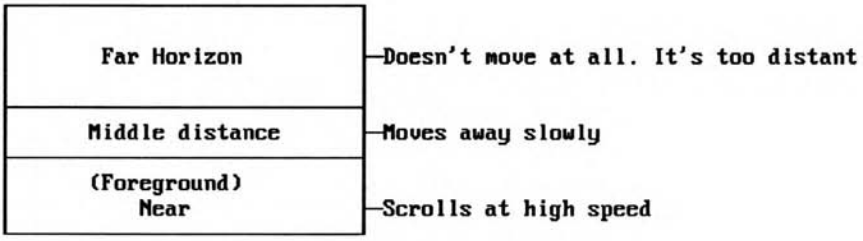

Figure 7.26: The principles of parallax scrolling

The scrolling is pretty straightforward, and can be accomplished with any one of my previous scrolling systems. The landscape can be drawn a single large map, just as it appears on the screen.

The coordinates of the map window can be changed immediately before each scrolling operation with MAP VIEW or TVIEW. You can then scroll each section using a separate call to the appropriate scrolling routine.

An example of this technique can be seen from the following program fragment. This demonstrates how the layers could be scrolled using coarse scrolling.

```
Rem Scroll near distance
Rem Set up drawing area on the invisible map buffer
Screen 1
TX=0:TY=150:BX=320:BY=200: Gosub TVIEW: Rem (Map View TX,TY to BX,BY)
Rem Update map
Add M1X,SP
Rem in an alternative system, you'd also need to change STY and SBY
Rem at this point
Screen Copy 1,TX+TWIDTH,TY,BX,BY To 1,TX,TY
Gosub TRIGHT:Rem (Map Right M1X,M1Y)
Scroll middle distance
Rem Reset drawing area
TY=120:BY=150:Gosub TVIEW: Rem (Map View TX,TY To BX,BY)
Rem Update map
Rem Don't forget about STY and SBY if you're using a different
Rem scrolling routine. These hold the coordinates of the area on
Rem the viewport which will be loaded with the new image
```

```
Screen Copy 1,TX+TWIDTH,TY,BX,BY To 1,TX,TY
Rem Twice the distance so half the speed
Add M2X,SP/2:Gosub TRIGHT: Rem Map Right M2X,M1Y)
```

After you'd scrolled the map buffer, you'd then copy both layers of the new screen into the viewport like so:

```
Screen 0 : Screen Copy 1,TX,TY,BX,20 To Logic(0),TX,TY
Screen Swap : Wait Vbl
```

Now for a simple example of this system in action. Just for a change, I'll use a fine scroller instead.

Example 7.12: Parallax Scrolling

```
Rem You may recognise bits of this program.
Rem It's a version of my MESSAGE rotator
Rem Save some memory
Close Editor
Close Workbench
Rem Create destination screen
Screen Open 0,320,200,8,Lowres
Double Buffer
Curs Off : Cls 0 : Paper 0 : Hide
Rem create source screen
Screen Open 1,680,100,8,Lowres
Curs Off : Cls 0 : Flash Off : Paper 0
Rem move destination screen in front of source
Screen To Front 0
Rem create a banner on the source screen
LANDSCAPE[100]
Screen 0 : Flash Off : Locate 0,0 : Pen 3 : Centre " Parallax Scrolling"
Rem scroll the screen
SPEED=2
Do
        Rem Rotate middle distance of screen 1
        Screen Copy 1,0,0,SPEED/2,50 To 1,640,0
        Screen Copy 1,SPEED/2,0,SPEED/2+640,50 To 1,0,0
        Rem Rotate near distance of screen 1
        Screen Copy 1,0,50,SPEED,100 To 1,640,50
        Screen Copy 1,SPEED,50,SPEED+640,100 To 1,0,50
        Rem copy rotated image of map buffer from screen 1 to 0
        Screen Copy 1,0,0,320,100 To 0,0,100
        Rem Switch logical and physic screens
        Screen Swap
        Wait Vbl
Loop
Procedure LANDSCAPE[N]
   Rem Create middle distance
   Rem I'm using a series of coloured triangles
   For B=1 To N
   C=Rnd(Screen Colour-2)+1 : X=Rnd(640) : Y=49 : W=Rnd(40)+2 : H=Rnd(40)+9
   Ink C : Polygon X,Y To X+W,Y To X+W/2,Y-H To X,Y
   Next B
   Rem Create near background
   Ink Screen Colour-1
   For L=1 To Screen Width-1 Step 32
   Draw L,100 To L+31,51
   Next L
End Proc
```

7.6.2 Dual playfield

The previous system is very effective, but it only works if the two scrolling zones are completely separate. Sometimes however, this just isn't possible. Supposing your landscape consisted of a series of tall narrow buildings such as skyscrapers. The buildings nearest the player would totally obscure the ones in the background. But if there were any gaps between the foreground buildings, the skyscrapers behind them would be visible in the distance.

If we want to handle these landscapes, we'll need to exploit a curious command called DUAL PLAYFIELD. This takes two AMOS Basic screens and combines them to form a single composite display:

`Dual Playfield FOREGROUND_SCREEN,BACKGROUND_SCREEN`

foreground_screen and *background_screen* are the identification numbers of two perfectly normal AMOS Basic screens which we've created previously. These can display anything up to eight colours, and can be double buffered and scrolled in the usual way.

When we call this command, the foreground screen will positioned directly over the background screen. The clever thing about dual playfields though, is that the foreground screen treats colour zero as if it were transparent. So any areas with colour zero will be automatically replaced with the appropriate sections of the background screen. Imagine the two screens were displayed on two pieces of transparent plastic or cellophane. If we placed the foreground screen on top of the background, the image it contained would be almost totally hidden from view. The only sections which would be visible, would be those which showed through the transparent, unshaded areas of the foreground.

Dual playfield screens provide us with the perfect way of generating terrific parallax scrolling effects in our games. They can be defined using the following procedure:

- Create two AMOS Basic screens using the SCREEN OPEN command. These should have the same resolution (Lowres or Hires), and a maximum of eight colours. (4 for Hires).
- If required, activate double buffering as normal.
- Add a call to DUAL PLAYFIELD using the appropriate screen numbers.

```
Screen Open 1,320,200,8,Lowres
Double Buffer
Rem the two screens DON'T have to be the same size!
Screen Open 2,640,100,8,Lowres
Double Buffer
Dual Playfield 1,2
```

- You can now scroll each screen using any one of my previous scrolling routines. Each screen is stored in memory completely separately. So you can select the screen with the SCREEN command, and scroll your images using:

```
Rem Scroll foreground
Screen 1
IX=IX1: Gosub SRIGHT
Rem IX1 holds the position of the viewport on foreground screen
Add IX1,SP1:Rem SP1 sets the scrolling speed of the foreground
Rem Scroll background
Screen 2
Rem IX2 holds the position of the viewport on background screen
IX=IX2: Gosub SRIGHT
Add IX2,SP2 :Rem SP2 sets the scrolling speed of the background
```

The only restriction is with SCREEN OFFSET, which seems to generate odd results during horizontal scrolling of a dual playfield screen. It does however, work fine for vertical scrollers.

The foreground screen is special by the way, as it provides the colour values for both screens.

Colour numbers 0 to 7 are displayed on the foreground, whereas colours 8-15 are reserved for the background screen. Remember that colour zero is transparent. So any colour values you've assigned to this index will be ignored. Generally, these colour assignments will only be important when you are setting up the palette. If you are drawing onto the background screen, you can forget about this system completely, and use colour indexes from 0 to 7 as usual. These will be automatically converted into the appropriate foreground colour by AMOS Basic.

According the original AMOS Basic manual, you can only move the foreground screen with SCREEN OFFSET. Apparently this problem has been fixed in the latest versions of AMOS Basic, but it's worth mentioning that you can swap the foreground and background screens around using the DUAL PRIORITY instruction like so:

```
Dual Priority BACKGROUND,FOREGROUND
```

This reverses the display so that background screen is repositioned over the foreground.

DUAL PLAYFIELD is a strange instruction, and occasionally generates equally strange effects. The problems occur when it's used alongside certain AMOS commands such as VIEW. It's therefore sensible to save your programs carefully onto the disk before you make any major alterations to your routines. If you're unlucky, the results of your latest experiment could be disastrous! Personally, I've encountered everything from unwanted colour shifts, to stretched or distorted displays. So if something weird happens, try removing any recent changes to your program. It may NOT be your fault.

7.7 Conclusion

AMOS Basic can make smooth scrolling effects an achievable reality rather than just an impossible dream. With a little hard work, you should be able to add scrolling backgrounds to any of your own games. Often, the results will be indistinguishable from those in a commercial game. Xenon III here we come!

8

Animation Techniques

AMOS Basic provides us with a host of powerful instructions which make it very easy to generate fantastic animation effects. I'll now demonstrate how we can exploit some of these features to add a little extra spice to our games.

8.1 Animating a Bob or Sprite

One of the most common requirements is the ability to animate moving objects such as the aliens in an arcade game. As you would expect from AMOS Basic, there's a mouth-watering range of possibilities.

8.1.1 Anim revisited

I'll begin with a closer look at the AMAL Anim command, first encountered in Chapter 1. It provides a direct way of animating any Bob or sprite through a complex sequence of images. Now's the chance to discover some of the special tricks which make this command so versatile. To jog your memory, here's a quick description of the standard instruction.

`Anim count,(image1,delay1)(image2,delay2)..`

count sets the number of times the animation sequence will be repeated on the screen. If we use a zero in this position, the animation will continue indefinitely.

image1,image2: enter a series of images from the sprite bank which will be displayed by our object.

delay1,delay2: choose the time each image will be held on the screen. All delays are measured in units of a 50th of a second.

Remember, this is an AMAL instruction, not a Basic one, so it can only be executed from inside an AMAL command string. To use it, we'll first need to assign a Channel to our object, and then start up the animation with a separate AMAL instruction. The general technique is:

```
Rem Assign a channel
Channel 1 To Bob 1Rem Position object on screen
Bob 1,100,100,1
Rem Enter our command into AMAL
Amal 1,"Anim 0,(1,41)(2,4)(3,4)(4,4)"
Rem Start it running
Amal On 1
```

Here's a more practical example for you to get your teeth into:

```
Rem Load some images
Load "Extras:Sprite_600/Vehicle/Scooter.Abk"
Curs Off : Flash Off : Cls 0 : Hide On Inverse On
Centre "<Scooter mouse!>"
Get Sprite Palette
Rem Assign a channel to our object
Channel 1 To Sprite 8
Rem Move the object to its starting position
Sprite 8,X Mouse,Y Mouse,1
Rem Load commands into AMAL channel
Amal 1,"Anim 0,(1,5)(2,5)(3,5)(4,5)(5,5)"
Rem Execute AMAL channel number 1
Amal On 1
Do
  Rem Move sprite 8 to the current mouse coordinates
  Rem The comma at the end means: Use the existing image
  Rem This will be animated continually with our ANIM command
  Sprite 8,X Mouse,Y Mouse,
  Wait Vbl
Loop
```

Just to complicate matters, there's also a Basic form of the ANIM command. This was originally included in the ST version of AMOS Basic (STOS), and is only provided for compatibility purposes between the two systems. Since it's been completely superseded by AMAL, I won't examine it in any great detail. But just in case you encounter it in your travels, here's some general information.

ANIM c,"(image,delay),(image2,delay2),(image3,delay3)....

Animates an object assigned to channel number *c*.

As you can see, the animation string is very similar to the AMAL version, except that there's no *count* value at the beginning, and a single comma is required between each pair of images. We can, however, cycle through the sequence continuously by placing a simple L character after our commands:

```
Anim 1,"(2,1),(3,1),(4,1)L"
```

This repeatedly flicks the object assigned to channel 1 through the images 2, 3, and 4.

ANIM ON Starts the animation sequence running.

We can include an optional channel number to execute a single animation at a time.

ANIM OFF Aborts the animation.

Examples:

```
Rem Aliens
Rem Load some images from the EXTRAS disc
Load "extras:sprite_600/aliens/alien1.abk"
Rem Set up the screen
Curs Off : Flash Off : Cls 0
Double Buffer
Get Sprite Palette
Rem Assign AMAL Channel 1 to Bob 1
Channel 1 To Bob 1
Rem Position bob at the centre of the screen
Bob 1,160,100,1
Rem Define animation sequence
```

```
Anim 1,"(1,6)(2,6)(3,6)(4,6)(5,6)L"
Rem start animation


Anim On
Rem SUBMARINE
Rem Load some images


Rem Note odd spelling of SUBMARIN
Load "Extras:Sprite_600/Water/Submarin.Abk"
Rem Set up screen
Curs Off : Flash Off : Cls 0
Double Buffer
Get Sprite Palette
Rem Assign Channel 1 to bob 1


Channel 1 To Bob 1
Rem Position bob 1 (the bubbles!)
Bob 1,150,100,2
Rem Position Bob 2 (This is the submarine!)
Rem I think it's the Nautilus from 2000 leagues under the sea
Bob 2,100,110,1
Rem define animation string. Spot the L for loop!
Rem It's equivalent to Amal 1,"Anim 0,(2,5)(3,5)(4,5)(5,5)"
Anim 1,"(2,5)(3,5)(4,5)(5,5)L"
Rem Fire up the animation
Anim On
Wait Key
```

At the time, ANIM was considered a real innovation. But although it's slightly more compact than the AMAL system, it's also far less flexible. If you're still interested, see the AMOS manual for a full explanation.

8.1.2 Sprite_600

As always, the best way to familiarise ourselves with an instruction is to look at some practical examples. If we examine the AMOS Extras disk, we'll find a small program called SPRITE600.AMOS. Don't confuse it with the FOLDER SPRITE_600. This has a "*" character in front of it in all our directory listings. SPRITE600 contains dozens of useful animation effects, and includes neat demonstrations of all the animation sequences provided for us on the Extras disk. We can list these Anim definitions on the screen in the following way. First, load SPRITE600.AMOS into your Amiga, and bring up a listing of the program in the AMOS editor window. Now press the cursor down arrow until you see the lines:

Rem ----------
Rem ----* *Alien option* *----
Rem ----------
This is the start of a long list of DEMO procedures.
Procedure ALIEN1_DEMO
: :

Normally, all these procedure definitions are hidden neatly away in memory. We can, however, list them onto the screen by moving the cursor over the procedure name, and pressing the F9 key.

Once we've opened these procedures, we can now use the animation sequences directly in our own programs. But we will obviously need to initialise the screen, and set up our AMAL

animation channels first! See the SETUP, SETDOWN and SETCHAN procedures at the end of Sprite_600.AMOS for an example. The AMAL Anim command can also be used in conjunction with the movement tables I showed you in Chapter 3. This will require us to make a couple of minor changes to our movement routines.

- Set the Animation channels to the appropriate Blitter objects or Sprites with the CHANNEL command: `Channel N to Bob B`

  where *B* is the number of our Blitter object, and *N* is the animation channel we wish to assign to it.

- Load the animation sequence into an AMAL command string such as:

  ```
  C$="Anim 0,(1,5)(2,5)(3,5)"
  ```

- Before assigning this animation, we need to position each Bob outside the normal screen area: `Bob B,-100,-100,SHIP`

  B is the number of the Bob, and SHIP is the first image of our new animation sequence.

- We can now start the animation with: `Amal N,C$: Amal On`

- Finally, we should carefully remove the image numbers from the Bob or sprite commands we're using to move our objects on the screen. A typical movement command would be: `Bob B,X(P),Y(P),SHIP`

  This should be changed to: `Bob B,X(P),Y(P),` The "," just tells AMOS Basic to re-position the existing image, and prevents our movements interfering with the Animation sequence.

8.1.3 Animating a large object

Large objects, such as level guardians pose special problems in our games. Supposing we wanted to create a large, wiggling octopus on the screen. Assuming that each frame of our animation sequence used a single 128 by 128 image, we need around 8k of memory per image. So if we attempted to store each frame of our animation individually, we'd rapidly run out of space. The solution is to split our object up into several smaller components. The bulk of the octopus can now use a single massive image, and we can animate the various body parts using their own separate routines.

Let's have a go at splitting up our octopus. This could be divided into the following images:

The head: This would be very large, and would therefore use a single fixed image.

The eyes: These could be generated with a series of 16x16 images. We could animate these images to create a neat "rolling eyes" effect.

The tentacles: These would be long thin streamers and could be generated using a number of 64 by 20 images. We could even split each tentacle up into several still smaller segments to provide extra flexibility.

Note that this system would only work if we were displaying all our objects on the screen in a single operation, using either the UPDATE or BOB DRAW commands. Otherwise, the various limbs would move out of step with each other.

8.1.4 Creating an explosion

Explosion effects should be short, sweet, and preferably dramatic. This is especially important in an arcade game, as we don't want the explosion to interfere with the rest of our object movements. If we get it wrong, there's a real danger of creating a *dying swan* effect, and clogging up the action with useless explosions.

The speed of our explosion doesn't just depend on the number of images of course. It's also based on the amount of time they are actually displayed on the screen. Take the following example, which gradually increases the delay between successive images in an animation sequence.

Example 8.1: Death and destruction

```
Double Buffer
Hide On : Curs Off : Cls 0
Load "Extras:sprite_600/Space/Xplosion.Abk"
Flash off: Get Sprite Palette
Channel 1 To Bob 1
For X=1 To 10
  Bob 1,X Screen(X Mouse),Y Screen(Y Mouse),
  Amreg(0)=X
  Amal 1,"Anim 0,(1,RA)(2,RA)(3,RA)(4,RA)(5,RA)(6,RA)(7,RA)
  (8,RA)(9,RA)(10,RA)(11,RA)(12,RA)(13,RA)(14,RA)(15,RA)"
  Amal On
  Wait 10
  Locate 0,0 : Print "Explosion";X
  Locate 0,20 : Centre "<Hit a key" : Wait Key
Next X
Bob Off
```

The first couple of explosions are fine, but after that things tend to slow down a little. It's clearly a matter of opinion about which speed is best, but personally I prefer a speed value of around 2. Notice how I've substituted the variable *RA* for the delay value in the Anim command. This allowed me to change the speed of my animation sequences directly inside my program, with the `Amreg(0)=X` instruction. The above explosion uses a total 15 different images. In many games we could get away with far fewer. Here are a couple of explosions which use 11 and 6 images respectively.

Take Example 8.1 and replace the Load command with:

```
Load "Extras:sprite_600/aliens/alien4.Abk"
```

Now swap the ANIM commands with either:

```
Amal 1,"Anim 1,(9,RA)(10,RA)(11,RA)(12,RA)(13,RA)
(14,RA)(15,RA)(16,RA)(17,RA)(18,RA)(19,RA)"
```

or:

```
Amal 1,"Anim 1,(9,RA)(11,RA)(13,RA)(15,RA)(17,RA)(19,RA)"
```

Try using some of the other animation sequences from the Extras disk. Most of the image files in the ALIENS folder also have their own individual explosion effects towards the end. So there's plenty of scope for experimentation.

8.1.5 Animation using AMAL FOR..NEXT loops

If you've followed my earlier explanation of the AMAL animation system, you'll already know that AMAL keeps a running count of each object's coordinates in the registers X and Y. There is however, also an A register which holds the present image number. If we change this register inside our AMAL program, we can flick our object through a complex animation sequence. For example:

```
Let A=10
```

Immediately assigns image number 10 to the current object.

Let A=A+1
Displays the next image in a predefined animation sequence.

Here's an example:

Example 8.2: Animating an object with an AMAL FOR..NEXT loop

```
Double Buffer : Hide On : Curs Off : Cls 0
Load "extras:sprite_600/Space/Xplosion.Abk"
Flash Off : Get Sprite Palette
Channel 1 To Bob 1
  Bob 1,160,100,
  F$="Loop: For R0=1 To 15; Let A=R0; P; Next R0; Jump Loop"
  Amal 1,F$
  Amal On
Do
Loop
```

This system does have a couple of advantages over the standard ANIM command. If the animation sequence is longer than about 10 images, the AMAL command string can become pretty unwieldy. The equivalent FOR..NEXT loop is much simpler. Note the use of the AMAL P or Pause instruction. It's more or less equivalent to the AMOS WAIT VBL command. The more pauses you include before the Next, the longer the delay between successive images.

Another possibility is to animate our objects as part of an AMAL joystick handler. We can now give each movement direction its own separate image. So our ships appear to bank or dive as they are moved across the screen.

Whenever the user pulls the joystick, our current object can be assigned to a brand new image with an appropriate AMAL Let command. Here's a fragment of the AMAL joystick routine from Chapter 3 which demonstrates this technique.

```
Rem define default alien
A$=A$+"L: Let A=1;"
A$=A$+"If J1&1 then Jump E ;" : Rem up
A$=A$+"  If J1&2 then Jump F ;" : Rem down
A$=A$+"  If J1&4 then Jump G ;" : Rem left
A$=A$+"  If J1&8 then Jump H ;" : Rem right Rem Jump back to start
A$=A$+"Jump L;"
Rem UP
Rem Y holds the Y coordinate of the current object
Rem when it's changed, AMAL moves the object automatically
Rem If the object is already at the top of the screen
A$=A$+"E: Let Y=Y-RY;"
Rem Use image two when the ship moves UP.
Rem Other directions would display other image numbers
A$=A$+" Let A=2; "
A$=A$+"If Y>RU then Jump L;"
Rem This line reverses the movement
A$=A$+"Let Y=Y+RY; Jump L;"
 :  : :
handle other directions here
 :  :  :
```

8.1.6 Animation tables

If we run out of animation channels, we can also animate our objects directly from AMOS Basic. The general technique is as follows: First, create an standard AMOS array containing a list of images to be displayed in sequence.

```
Rem defines a table 4 images long. It could be MUCH longer.
Dim AN(4)
```

We now load it with the numbers of our images:

```
Rem NI holds the number of images
For A=0 to NI
   Read SHIP
   AN(A)=SHIP
Next A
Rem holds sequence to animate our object through images 4,3,2,1
Data 4,3,2,1
```

Once we've set up our table, we can display our image on the screen with:

```
Do
  Bob B,X,Y,AN(C)
  Add C,1,1 To NI
Loop
```

Here's a complete example for you to type in:

```
Dim AN(15)
Rem Thirteen images
NI=13
Rem set the delay between each image
DELAY=5
Rem Make up a simple animation table
For A=1 To NI
  AN(A)=A
Next A
Rem Load a whale from the extras disc
Load "Extras:Sprite_600/Water/whale.Abk"
Rem Set up screen
Double Buffer
Get Sprite Palette
Curs Off : Flash Off : Cls 0
Rem First image is one
A=1
Rem Main loop
Do
  Rem Display bob with new image
  Bob 1,120,110,AN(A)
  Rem Move to next image
  Add A,1,1 To NI
  Rem Wait a bit
  Wait DELAY
Loop
```

These tables only make sense if we are moving our objects by hand, as they require our program to keep permanent track of the objects we are currently animating. But if we are already using movement tables, we can often incorporate animation directly into our routines with practically no extra programming. We simply replace our existing Bob command with something like:

```
  Bob B,X(I),Y(I),AN(A):Add I,1,1 To N:Add A,1,1 To NI
```

Animation tables are also valuable for animating objects such as icons or screens, which just can't be handled using the standard AMAL commands.

8.2 Colour Scrolling

8.2.1 Basic principles

The Amiga allows us to use up to 32 colours on the screen at any one time (more in HAM or EXTRA HALF-BRIGHT modes). Each colour can be individually selected from a grand total of 4,096 possible shades. As I explained in Chapter 1, the values for these colours are stored in a set of internal variables known as the colour indexes. Whenever one of these is changed with the COLOUR or PALETTE commands, the appropriate screen colour is altered immediately.

AMOS Basic refers to each index using a simple number from 0 to 31. We can therefore choose the precise shade which will be displayed by any particular index with a simple call to the COLOUR instruction.

COLOUR index,$RGB

```
colour 1,$F00:rem load index one with $F00 (RED)
```

So far, so good. Let's delve a little deeper!

When we run an AMOS program, a standard screen is created for our graphics. Normally, this screen is filled with a large block of colour 1 (red). But what would happen if we loaded a different index with the same shade, and then used it to draw some new graphics directly on top of the original? Well, we can check this out by entering the following lines from direct mode:

```
colour 2,colour(1)
ink 2:bar 60,50 to 260,150
```

Since our bar uses the same colour as the background screen, it's completely invisible. However, as it's drawn in a different index, it's still stored totally separately. We can prove this by loading index 2 with a colour such as white:

```
colour 2,$fff
```

Suddenly, a white bar winks into view.

As you may know, the Amiga stores all our screens as a list of index numbers, with a single value for each individual point on the display. The basic idea can be seen in Figure 8.1, although the actual storage system is obviously rather more complicated.

```
11111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111
11111111111111222222222222222211111111111111
11111111111111222222222222222211111111111111
11111111111111222222222222222211111111111111
11111111111111222222222222222211111111111111
11111111111111222222222222222211111111111111
11111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111
```

Figure 8.1: Screen composition

In the previous example, our bar was drawn in index 2, and the background screen used index 1. But the actual colours assigned to these indexes are stored completely independently. So we can change them at a moment's notice, generating a startling range of animation effects.

Here's a larger example:

Example 8.3: A moving box

```
Curs Off : Cls 0 : Flash Off
Rem draw a series of colour blocks on the screen
For Y=0 To 2
  For X=0 To 4
    Rem The ink colour ranges from 1 to 15
    Ink Y*5+X+1
    Rem draw our block
    Bar X*60,Y*60 To(X+1)*60,(Y+1)*60
  Next X
Next Y
Locate 0,24 : Centre "<Press a key to move box>"
Wait Key
Locate 0,24 : Paper 0 : Cline
MOVE1:
Rem Set the movement speed
SPEED=5
Rem Move our blocks
Do
  Wait SPEED
  Rem Change the colour of each block in turn
  For C=1 To 15
    Rem Set the colour of the Last block to black
    Colour C-1,0
    Rem Set the colour to the current block to YELLOW
    Colour C,$FF0
    Wait SPEED
  Next C
Loop
```

This draws 15 rectangular boxes on the screen, each using a different colour index from 1 to 15. The program now takes our boxes, one at a time, and removes their immediate neighbour by zapping the associated colour index down to zero (black). It then loads the current box index with a value of $FF0 (yellow), and a yellow rectangle seems to spring into view against the black background. The result is the illusion of a single box jumping back and forth across the Amiga's screen. In reality though, all that's happening is that the colour of each box is flicking continually from black to yellow.

The same technique can be applied to animate any sequence of objects we like. Remember that when we change a colour, we affect all the objects painted with it. So if we want to animate an object against a colourful background, we will need to use separate colours for the object and the background areas. It's also vital to ensure that none of the objects overlap. Otherwise the animation effect simply won't work.

One limitation of Example 8.3, was that our program had to change all the colours by itself. AMOS Basic includes three useful instructions which allow us to perform the entire process automatically.

8.2.2 The FLASH command

The AMOS FLASH command takes a colour index and repeatedly flicks it through a whole series of colour values. The format is fairly similar to the AMAL Anim command.

`FLASH index,"(colour1,delay1)(colour2,delay2)...`

index is the number of a colour index from 0 to 31.

colour1,colour2 hold a list of values which will be assigned to our chosen colour. The numbers are entered in RGB notation, but unlike COLOUR, there's no need to include a $ sign at the beginning. We DO however need to enter all three digits, even if they're set to zero! So 000 is ok, but 0 generates an error.

delay sets the length of time each colour in the sequence will be displayed on the screen. Like Anim, it's measured in units of a 50th of a second. For example:

```
ink 2: bar 60,50 to 260,150
flash 2,(fff,10)(000,10)"
flash 2,"(fff,6)(f00,6)(0f0,6)(00f,6)"
flash 2,"(f00,3)(0f0,3)(00f,3)
flash off
```

Supposing we were displaying a picture of a computer bank. Each button on the control panel could now be made to flash through several colours. So as we looked at the computer, the display would appear to be continually changing in front our eyes. Since FLASH works independently of our Basic programs, once we have initialised the display, we can forget about the animation completely. We could also use this command to create the alert screens found in some games. Here's another example:

```
flash 0,"(f00,3)(0f0,3)(00f,3)
Rem Ouch!
wait key
flash off
```

8.2.3 The SHIFT UP/SHIFT DOWN commands

The SHIFT UP and SHIFT DOWN commands take our existing colour values and rotate them one at a time through a selected group of colour indexes.

SHIFT UP delay,first,last,type

SHIFT UP copies the contents of each colour index directly into the one in front. So:

Colour(first+1)=Colour(first)
Colour(first+2)=Colour(first+1)
: :
: :
Colour(last)=Colour(last-1)

first and *last* choose the range of colour indexes which will be rotated by our instruction.
delay sets the delay in 50ths of a second between each colour shift.

The whole process is completely automatic, and will be executed quietly in the background by the AMOS system without affecting the running of our existing AMOS programs. Just like AMAL in fact!

type selects one of two options. A value of 1 indicates that the contents of *last* will be loaded straight into *first* before it's changed. So although the colours rotate continuously on the screen, our original colour values remain intact. They're merely shuffled through the available indexes. On the other hand, if we use a zero in this position, AMOS will completely discard the colour value of index *last*. After a while, all the colour indexes will therefore contain an exact copy of the first colour. Let's have a look at a practical example of the command.

Example 8.4: SHIFT in action

```
Close Editor
Screen Open 1,640,30,4,Hires
Screen Display 1,,250,,
Curs Off : Hide On
Palette 0,$F,$FFF
Pen 2
Centre "<You'll need to wait a while before it gets interesting>"
Screen 0
Cls 0
DELAY=2
Flash Off : Rem try removing it for some terrific effects!
Degree
Rem Set BASE to $1,$110,$011,$010,or $100 and convert the colour
Rem Palette to shades of a single colour
Rem Try removing the next rem
Rem BASE=$111 : For C=1 To 15 : Colour C,C*BASE : Next C
For I=1 To 180 Step 1
  Rem cycle colours from 1 to 15 as the objects are drawn
  B=(I mod 15)+1
  Ink B
  Draw 160,100 To 160+100*Sin(I*2),100+100*Cos(I*2)
  Rem alternative effects
  Rem remove as many, or as few rems as you like
  Rem Box 160-I,100-I/2 To 160+I,100+I/2
  Rem Polyline 160,100-I To 160+I,100 To 160,100+I To 160-I,100 To
  160,100-I
  Rem Circle 160,100,I
Next I
Screen 1
Locate 0,0 : Cline
Centre "<Change the speed and direction with the arrow keys>"
D$="up"
Screen 0
Shift Up DELAY,1,15,1
Do
  X$=Inkey$
  If X$=Cup$ Then Bell 10 : Shift Up DELAY,1,15,1 : D$="up"
  If X$=Cdown$ Then Bell 11 : Shift Down DELAY,1,15,1 : D$="down"
  If X$=Cleft$ Then Bell 1 : Add DELAY,-1,0 To 100 : SP=1
  If X$=Cright$ Then Bell 5 : Add DELAY,1,0 To 100 : SP=1
  If SP=1
    If D$="down"
      Shift Down DELAY,1,15,1
    Else
      Shift Up DELAY,1,15,1
    End If
  End If
  If X$<>""
    Screen 1
    Locate 0,1 : Print "Direction ";D$;" ";
    Locate 0,2 : Print "Delay ";DELAY;"  ";
    Screen 0
  End If
  SP=0
Loop
```

When we run this program, a rotating circular wheel appears, composed of an attractive arrangement of coloured spokes. Actually though, the spokes aren't moving anywhere. That's

because I've sneakily arranged it so the colour indexes used by the lines repeat continuously from 1 to 15. This is accomplished with the instructions:

```
B=(I Mod 15)+1
Ink B
```

So the first line is drawn in colour 1, the second in colour two and so on. When we rotate our colour values, each line is rapidly loaded with the colour value of its immediate neighbour. It now appears as if the line has rotated anti-clockwise, whereas it's still fixed firmly in its original position. The only thing that's really changed is its colour. We can demonstrate this process by adjusting the delay between each rotation step. In my current example, the left arrow reduces the delay, and the right arrow increases it. Try changing the delay to something like 50. You'll now be able to see each line changing colour approximately once a second. Also have a look at some of the other rotation sequences I've provided. These can be accessed by simply removing the rems below the DRAW command.

Before continuing, don't forget remove the REM from the FLASH OFF instruction at the start of the program. Since colour number 3 is now continually changing, the colours slowly shift through a vast number of possible shades. Whoosh!

We can also incorporate this system directly into Example 8.3. Try replacing the MOVE1: routine with the following lines:

```
MOVE1:
SPEED=5
For C=0 To 15 : Colour C,0 : Next C
Colour 1,$FF0
Colour 15,0
Shift Up SPEED,1,15,1
Wait Key
```

This performs the whole process automatically and allows our program to get on with something more interesting instead.

SHIFT DOWN delay,first colour,last colour,type

As you'd expect, SHIFT DOWN is identical to the previous SHIFT UP instruction, except that all the colours are rotated in the opposite direction. The colour values are shifted in the following order.

Colour(first)=Colour(first+1)
Colour(first+1)=Colour(first+2)
: :
: :
Colour(last-1)=.Ccolour(last)
Colour(last)=Colour(first) Only set if TYPE=0

You don't have to look too far for an example of this command, as I've already included it in Example 8.3. The up arrow selects SHIFT UP and the down arrow chooses SHIFT DOWN. Easy eh?

SHIFT OFF

This stops any colour rotations, and fixes the colours in their current values.

These SHIFT commands are perfect for creating the hyperspace sequences found in games like Captain Blood. A full demonstration can be found the hyperspace program at the end of this chapter.

8.2.4 The FADE instruction

FADE allows us to progressively fade a group of colour indexes from one set of values to another. The most common use of this is to fade out the background during our game's initialisation phase.

Normally, all the colours on the screen are faded to black while our new game screen is being drawn. The colour palette can then restored as the game commences, producing a highly professional fading effect.

There are four available options:

`FADE speed`
Fades all the colour values down to zero. It's a total black-out!

`FADE speed,colour1,colour2,colour3..`
Smoothly converts all the screen colours to the values specified in the instruction. Any values we miss out will be ignored.

`FADE speed TO s`
Fades all the colour values used by screen *s* into the current colour indexes.

`FADE speed TO -1`
Fades the colours used by the sprite images straight into the present screen.

FADE is not just limited to our loading screens however. It's actually an extremely versatile command, capable of smoothly scrolling any set of colours through several intermediate shades. Try entering the statement:

```
Fade 10,$F00
```

This fades the background colour down to red. The *speed* indicates the interval in 50ths of a second between each step, and can vary between 1 and 999. All fades are performed in exactly 15 steps. So the maximum time to completion is about 280 seconds (999*15/50). Here are some examples

```
Rem Fading the image colours into the screen
Load "Extras:sprite_600/flight/planes.abk"
Fade 10
Wait 150
Cls 0 : Flash Off
For I=0 To 7
  Bob I,I*32,I*16,I
Next I
Fade 10 To -1
Wait 10
Rem Fading a particular colour
Curs Off
Locate 0,10 : Centre "I'm fading out!"
Fade 10,,$FFF,$A40
Wait 150
Fade 10,,,0
Wait 150
Fade 5,,$A40
Wait 75
Fade 5,,,$FFF
Wait 75
```

Notice the commas between the values in our FADE command. These allow us to fade some colours without affecting the rest.

It's also possible to fade away any particular Blitter object or sprite. Providing we keep the colours used by our objects completely separate from the rest of the screen, we can teleport them away with just a single FADE instruction.

```
Rem fading between two screens
Close Workbench
Close Editor
Hide On
Rem Load a title screen
Load Iff "amos_data:iff/amospic.iff",0
Rem fade it away
Fade 10
Rem wait for the fade to complete
Wait 150
Screen Open 0,320,210,16,Lowres
Rem Let's fade in a blitter object, just for a change
Rem This is a digitised, and slightly adjusted version of
Rem Richard Vanner, the original AMOS Project Manager
Rem Its about time he got a mention in one of my books!
Rem Load the sprite image
Load "amos_data:sprites/Count_Dickular_Sprite.abk"
Cls 0
Flash Off
Rem Position a strange face in the centre of the screen
Bob 1,30,30,1
Rem Fade the picture slowly into view
Rem The result is rather reminiscent of DR WHO
Fade 10 To -1
Rem wait for the fade to complete
Wait 150
Wait Key
```

8.2.5 Rainbows

All the previous colour commands made use of the same basic trick. They took a colour index and gradually adjusted its value over a period of time. But what would happen if we took the same colour index, and altered its contents according to its position on the screen? We'd now be able to re-use a single colour index to display hundreds of colours at a time, So a boring two-colour screen could be transformed into a fantastic rainbow effect! Impressed? Just wait until you see it in action!

Before we can create one of these rainbows, we first need to do a little initialisation work with the SET RAINBOW command.

SET RAINBOW n,index,height,red$,green$,blue$

n is the number of the rainbow from 0 to 3. We can therefore use a maximum of four rainbows at a time.

index is the number of the colour index we wish to change. We're only permitted to use indexes from 0 to 15.

height sets the height of the rainbow in lines. This may be much greater than the visible screen area, but only the first 280 lines or so can be displayed at once.

Finally, there's the *red$,blue$,green$* strings. These allow us to define the appearance of the rainbow using a series of special string commands. That's the theory anyway. The practice though, is that nobody actually uses these commands! Most people set the strings to blank values, and create the rainbow using a simpler system. So the standard SET RAINBOW instruction boils down to:

SET RAINBOW n,index,height,"","",""

When Francois Lionette was designing the RAINBOW instruction, he casually included an inconspicuous little function called RAIN.

RAIN(n,line)=shade

This sets the colour of any individual rainbow band to any one of 4,096 shades.

line is the number of the band we wish to define in rainbow *n*. Possible values range from 0 to the original height.

shade is a colour value from 0 to 4095 which will be loaded into our index when it is displayed on the appropriate point on the screen.

The easiest way to demonstrate how this works is with an example:

```
Rem Define the rainbow using index 1
Set Rainbow 0,1,200,"","",""
Rem Display the rainbow on the screen
Rainbow 0,0,49,200
Rem load each rainbow band with a different colour
For R=0 To 199
  Rem Band R=Colour value R
  Rain(0,R)=R
Next R
```

This creates a simple rainbow using colour 1 and positions it over the entire screen area. The first line defines rainbow number zero, using colour 1 and sets up 200 colour bands.

```
Set Rainbow 0,1,200,"","",""
Rainbow 0,0,49,200
```

RAINBOW is a new instruction which displays our rainbow on the screen, and moves it to the top line of the picture.

We now load each rainbow line with a colour value from 0 to 199, depending on its position. This results in a series of attractive bands on the screen. Enter direct mode and move the command window with the up and down arrow keys. As you can see, the rainbow is not limited to the current screen. It actually affects the entire display!

Now for another example which generates the same results on a pitiful two-colour screen.

```
Rem Open a two-colour screen
Screen Open 0,330,256,2,Lowres
Rem and define our rainbow as before
Set Rainbow 0,0,256,"","",""
Rainbow 0,0,49,256
For R=0 To 255
Rain(0,R)=R
Next R
```

This screen requires just 8k of memory. And the rainbow uses less than 500 bytes!

Here's a small experimenter for you to play around with:

```
Rem Open up a two colour screen
Screen Open 0,330,256,2,Lowres
Rem Set up rainbow
Set Rainbow 0,0,256,"","",""
Rem Initialise the screen
Curs Off : Cls 0
Hide On
Locate 0,2 : Centre "<Rainbow Experimenter>"
Rem Display rainbow
Rainbow 0,0,49,256
Rem Main loop
Do
  Rem Enter some values
  Locate 10,16 : Cline : Input "Starting colour?";CBASE
  Locate 10,17 : Cline : Input "Step size   ?";S
  Rem Check if maximum colour value will be greater than 4096
  If CBASE+255*S>4096
    Rem Error!
    Locate 7,24 : Print "Try some smaller values"
    Wait 50
    Locate 7,24 : Cline
  Else
    Rem Create rainbow
    For R=0 To 255
      Rain(0,R)=R*S+CBASE
    Next R
    Rem Redraw our new rainbow on the screen
    Rainbow 0,0,49,256
  End If
Loop
```

The starting colour sets the shade used by the first line in our rainbow, and the step-size changes the difference between each colour value on the screen. The starting colour can be in either decimal (normal) or $RGB format. So it's easy to set the start colour to any shade you wish. Feel free to type away at random. If you enter something silly, the program will ask you to try again!

Note: The experimenter program only demonstrates a fraction of the possibilities! We can actually design our rainbows directly on the screen using the fantastic Rainbow Warrior program. This is available from the AMOS PD Library for around £3, and is highly recommended! Here's the current address:

Sandra Sharkey
AMOS PDL
25 Park Road
Wigan
WN6 7AA

We can also use the RAINBOW command to move these rainbows around on the screen. RAINBOW takes the current definition and displays it at the appropriate position on the screen.

RAINBOW n,start,hy,height

n is the identification number of the rainbow we've created with our previous SET RAINBOW and RAIN instructions.

start is the number of the first rainbow line which will be displayed on the screen. If it's zero, the initial index colour will be taken from RAIN(n,0). And if, say, the start is 5, the first colour band will be RAIN(n,5).

hy sets the vertical position of the rainbow on our TV display. It's entered using those horrible hardware coordinates, and ranges from 40 (top of the screen) to around 312 (bottom).
height holds the height of our rainbow in lines. Each line is exactly one pixel tall. So we can fit a maximum of about 280 lines on our TV screen.

We can move our rainbow in two ways. One option is to create a massive rainbow, and scroll it through the screen by changing the value of start. Here's an example:

```
SPEED=2:Rem Set's the scrolling speed
CSTEP=16:rem sets the colours used to create the rainbow
RSTEP=2 :rem change to generate a whole family of effects
Rem open the screen
Screen Open 0,330,300,2,Lowres
Rem define the rainbow
Set Rainbow 0,0,4096,"","",""
Curs Off : Cls 0
Hide On
Rainbow 0,0,40,265
For R=0 To 255
  Rem This is an alternative random colour selector
  Rem B=Rnd(4096) : For C=0 To 15 : Rain(0,R*15+C)=B+C : Next C
  For C=0 To 15 : Rain(0,R*15+C)=R*RSTEP+C*CSTEP : Next C
Next R
For S=0 To 4097-255 Step SPEED
  Rainbow 0,S,40,265
  Wait Vbl
Next S
```

The core of this routine, is the bit at the bottom:

```
For S=0 To 4097-255 Step SPEED
  Rainbow 0,S,40,265
  Wait Vbl
Next S
```

This uses the RAINBOW command to scroll our colour values slowly through the screen, starting at value *S*. Note that *SPEED* sets the speed of the movement, whereas *CSTEP* and *BSTEP* just change the colours assigned to each rainbow line. Try changing them to generate a range of interesting effects. Another option, is to alter the position of the rainbow on the screen by changing *hy*:

```
Rem Open screen
Screen Open 0,330,255,2,Lowres
Rem Define rainbow
Set Rainbow 0,0,256,"","",""
Rem Set up screen
Curs Off : Cls 0
Hide On
SPEED=2:Rem Choose movement speed
Rainbow 0,0,40,255
Rem Load colour values into rainbow
For R=0 To 255
  Rain(0,R)=R*15
Next R
Do
  Rem Move rainbow down
  For S=40 To 312 Step SPEED
    Rem Redraw rainbow one place downwards
    Rainbow 0,0,S,255
    Wait Vbl
```

```
  Next S
  Rem Move rainbow back up again
  For S=312 To 40 Step -SPEED
    Rainbow 0,0,S,255
    Wait Vbl
  Next S
Loop
```

So far, all these routines have required us to move our rainbows directly inside our Basic program. However, there is also an AMAL version of these commands. This is activated with:

```
CHANNEL c TO RAINBOW n
```

We can now use the Move and Let instructions to move our rainbow smoothly across the screen. AMAL controls our rainbow in the following way:

The X coordinate of our object holds the start of the rainbow colours from the beginning of the RAIN table. And the Y coordinate sets the position of our rainbow on the screen. Here are a couple of examples:

```
Rem Scrolling a rainbow
SPEED=2:Rem Set's the scrolling speed
CSTEP=16:rem sets the colours used to create the rainbow
RSTEP=2 :rem change to generate a whole family of effects
 Rem Define a two colour screen
Screen Open 0,330,300,2,Lowres
Rem Create rainbow
Set Rainbow 0,0,4096,"","",""
Curs Off : Cls 0
Hide On
Rem Display rainbow
Rainbow 0,0,40,265
Rem load a different colour into each band
For R=0 To 255
  Rem B=Rnd(4096) : For C=0 To 15 : Rain(0,R*15+C)=B+C : Next C
  For C=0 To 15 : Rain(0,R*15+C)=R*RSTEP+C*CSTEP : Next C
Next R
  Assign an AMAL channel to our rainbow
Channel 1 To Rainbow 0
Rem 3842 is just 4097-256. 1921 is half this value.
Rem so each step is exactly two units wide
Rem See how the X coordinate controls the starting colour
Rem of our rainbow
R$="Move 3842,0,1921"
Rem Load our AMAL program into Channel 1
Amal 1,R$
Rem Execute it!
Amal On 1
Wait Key
Stop

Rem Moving a rainbow
Screen Open 0,330,255,2,Lowres
Set Rainbow 0,0,256,"","",""
Curs Off : Cls 0
Hide On
SPEED=2
Rainbow 0,0,40,255
For R=0 To 255
  Rain(0,R)=R*15
```

```
Next R
Channel 1 To Rainbow 0
R$="Loop: Move 0,256,64; Move 0,-256,64; Jump Loop"
Amal 1,R$
Amal On 1
Wait Key
```

As you can see, rainbows are terrific fun! What's more, they can transform a boring two-colour screen out of all recognition, using tiny amounts of memory. Experiment away!

8.3 Screen Animation

We'll now have a look at the various techniques which will allow us to animate entire AMOS screens.

8.3.1 Bouncy bouncy!

We've just seen how AMAL could be used to generate fantastic moving rainbows. But we can also use the same systems to bounce an entire screen around the display. This is ideal for demos and is extremely easy to produce.

As always, we first call up the AMOS channel command like so:

```
CHANNEL c TO SCREEN DISPLAY s
```

Channel now assigns channel number *c* to screen *s*. This screen can be in any mode we like. Providing it exists when our AMAL program is executed, all will be well. Once we've assigned our channel, we can move the screen using the familiar AMAL Move commands, or from a standard AMAL FOR..NEXT loop. The X coordinate now represents the position of our screen on the display, using the hardware system we saw in Chapter 1. This can be changed to a value from 112 to 448, but the screen will only be moved when the coordinate is an exact multiple of 16.

Also note that the Amiga's hardware places real limits on the display position. So if the X coordinate is outside the permitted range, the screen will break up in a most alarming manner. There's no need to worry however! When we move our screen back to a more sensible location, it reappears as normal.

The Y coordinate allows us to move the screen smoothly up or down at will. It can be set to any value from 0 to about 311. Without any more ado, here's an example:

```
Rem Load up a picture from the AMOS DATA disc
Load Iff "AMOS_data:IFF/amospic.iff",0
Rem Assign the positioning of the screen to AMAL channel 0
Channel 0 To Screen Display 0
Rem Horizontal movement
Rem S$="Loop: Move 256,0,16; Move -256,0,16; Jump L"
Rem Vertical movement
S$="Loop: Move 0,256,128; Move 0,-256,128; Jump L"
Rem Vertical and horizontal movement
Rem S$="Loop: Move 256,128,16; Move -256,-128,16; Jump L"
Rem Execute our AMAL program
Amal 0,S$
Rem Action
Amal On
Wait Key
```

We can also scroll our screen. The scrolling method uses the SCREEN OFFSET system we looked at in Chapter 7. SCREEN OFFSET may be a bit impracticable in a real game, but it's

great for demos. We simply draw our images on an extra large screen and assign this screen to an AMAL channel like so:

```
CHANNEL c TO SCREEN OFFSET s
```

Allocates channel *c* to screen *s*. The X and Y coordinate now set the starting point of the area to be displayed on our screen. By changing these coordinates we can smoothly scroll the entire picture through the screen.

```
Rem Open a wide, but narrow screen
Screen Open 1,960,10,16,Lowres
Rem Display the first 300 pixels
Screen Display 1,142,300,320,
Rem Load up a message string. This can be anything you like up to
Rem about 120 characters
T$="Here's an example of screen offset scrolling using AMAL. "
T$=T$+"Care of the Game Maker's manual By Stephen Hill "
Rem Display the text on the screen
Rem TEXT BASE just positions the text at the bottom of the line
Text 0,Text Base,T$
Rem Now load a picture from the AMOS DATA disc into screen 0
Load Iff "AMOS_data:IFF/amospic.iff",0
Rem Assign this to AMAL channel 0
Channel 0 To Screen Display 0
Rem And move it with an AMAL move command
S$="Loop: Move 0,256,128; Move 0,-256,128; Jump L"
Amal 0,S$
Rem Bring our text screen to the front of the display
Screen To Front 1
Rem Assign channel 1 to scrolling it
Channel 1 To Screen Offset 1
Rem And channel 2 for moving it
Channel 2 To Screen Display 1
Rem Create a string to scroll our screen from left to right
R$="Loop: Move 600,0,300; Move -600,0,300; Jump Loop"
Rem Create a command string to bounce our text up and down
D$="Loop: Move 0,-255,255; Move 0,255,244; Jump Loop"
Rem Action!
Amal 1,R$ : Amal 2,D$
Amal On
Wait Key
```

Ok, so it's all fairly useless stuff. But at least the scrolling's smooth! Let's return to more practical matters, with a look at screen flipping.

8.3.2 Screen flipping

This is an invaluable technique which can substantially improve the smoothness of our animation effects. The problem with the normal screen commands is that they allow us to see the drawing operations while they are taking place on the screen. Although most of these operations perform remarkably quickly, there's still a noticeable flicker to our animated graphics.

This can be seen from the following demo program which draws an expanding circle on the screen.

```
Rem Expanding circle
Close Workbench
Close Editor
Screen Open 0,320,220,16,Lowres
Paper 0 : Curs Off : Cls 0
Print "Normal  System"
Flash Off
SP=5
Timer=0
For B=SP*2+1 To 99 Step SP
  Rem Erase last circle
  Ink 0:Circle 160,120,B-SP
  Rem draw a new one
  Ink 2:Circle 160,120,B
Next B
Print "Time in seconds ";Timer/50.0
```

Yuck! Fortunately, there's a solution at hand. As I showed you in Chapter 7, we simply draw all our objects onto a separate logical screen which is invisible to the user. After the drawing has been completed, this screen can be swapped with the physical screen to produce a smooth, flicker-free animation. We can activate this system with the DOUBLE BUFFER command. AMOS will now automatically create our graphics in the correct screens, and swap between them when required.

Well, that's the theory anyway. Let's have a look at how it works in practice.

```
Rem Autoback in action
Close Workbench
Close Editor
Screen Open 0,320,220,16,Lowres
Paper 0 : Curs Off : Cls 0
Locate 0,0 : Print "Autoback  System"
Double Buffer
Timer=0
For B=SP*2+1 To 99 Step SP
  Ink 0
  Circle 160,120,B-SP
  Ink 2
  Circle 160,120,B
Next B
Print "Time in seconds ";Timer/50.0
Wait Key
```

Ok, so maybe the flicker's gone, but there's a massive slow-down in all our drawing operations. That's because AMOS is now having to work flat out to avoid the screen synchronisation problems we encountered in Chapter 7. In most cases, it's faster to put AMOS out of its misery and take control over the entire process ourselves. This can be accomplished with the AUTOBACK 0 command. Here's some pseudo-code which highlights the procedure:

```
create a separate logical screen with DOUBLE BUFFER
turn off the autoback system with AUTOBACK 0
Do
  Draw our graphics on logical screen
  swap the logical and physical screens with SCREEN SWAP
  wait for the screens to swap with a WAIT VBL
Loop
```

In an actual program, it would look rather like this:

```
Close Workbench
Close Editor
Screen Open 0,320,220,16,Lowres
Paper 0 : Curs Off : Cls :Flash Off
Locate 0,0 : Print "Screen switching system"
Rem Activate double buffering
Double Buffer
Rem Turn off the standard drawing system
Autoback 0
Timer=0
For B=SP*2+1 To 99 Step SP
  Ink 0
  Rem Handle screen synchronization
  Circle 160,120,B-SP*2
  Ink 2
  Circle 160,120,B
  Rem Switch between logical and physical screens
  Screen Swap : Wait Vbl
Next B
Screen Copy Physic To Logic
Screen Swap : Wait Vbl
Autoback 2
Print "Time in seconds ";Timer/50.0
Wait Key
```

As you can see, I've had to alter the drawing system slightly in order to keep the logical and physic screens in step with each other:

```
  Ink 0
  Rem Handle screen synchronisation
  Circle 160,120,B-SP*2
```

The `B-SP*2` bit gets the size of the circle to be removed from the screen. Since we are switching continually between two separate screens, each will only be redrawn once every two loops. So the last circle in the logical screen will be exactly two steps away from the new one. If all this seems hopelessly complicated you'll be pleased to hear that there's a neat little compromise we can make which simplifies the whole process enormously. After we switch our between the logical and physical screens, we perform the following command:

```
Screen Copy Physic To Logic
```

This copies the new physical screen into the new logical screen and ensures that both screens contain identical images. Our two screens are now perfectly synchronised so there's no need to bother with any fancy programming tricks. We just display all our graphics as normal, and forget about screen synchronisation completely.

Here's a final example which demonstrates this technique.

```
Close Workbench
Close Editor
Screen Open 0,320,220,16,Lowres
Paper 0 : Curs Off : Cls 0
Flash off
Rem fire up double buffering
Double Buffer
Locate 0,0 : Print "Direct Synchronisation"
Rem turn off autoback system
Autoback 0
Timer=0
For B=SP*2+1 To 99 Step SP
```

```
  Rem copy new physical screen into new logical screen
  Rem so both screens are exactly the same
  Screen Copy Physic To Logic
  Ink 0
  Rem There's no need to worry about synchronisation
  Circle 160,120,B-SP
  Ink 2
  Circle 160,120,B
  Screen Swap : Wait Vbl
Next B
Screen Copy Physic To Logic
Screen Swap : Wait Vbl
Autoback 2
Print "Time in seconds ";Timer/50.0
```

Note that if we were completely redrawing our screens during each loop, we could replace our SCREEN COPY command with just CLS 0. That's the approach used by the forthcoming AMOS 3D package!

8.3.3 Dissolving between two screens

AMOS Basic allows us to define separate screens to hold the various displays needed for our programs. So we can have one screen for our high score table, another for our title screen, and yet a third for our main playing area. We've already seen how we could switch between these screens using the FADE command. But FADE only scratches the surface of AMOS Basic's abilities.

APPEAR

I'll begin with a brief look at the APPEAR command. This provides a simple way of dissolving the current display between any two pictures.

APPEAR source TO destination,effect

APPEAR slowly transforms the contents of the destination screen into an exact copy of the source screen. The two screens should be of the same type and size, and contain the same colour values. Otherwise the effect might be rather strange!

source is the number of the screen from which the new image will be loaded.

destination is the screen on which the source image will appear.

effect holds a number from 1 to 32000 which chooses the type of fade. Some effects convert the destination screen to an exact copy of the source image, whereas others only copy over selected parts. Unfortunately, there's no easy way of visualising the results of any particular effect number in advance. We have to try it out on the screen and observe the fade for ourselves. Here's a small experimenter program for you to play around with:

```
Rem Appear demo!
Close Workbench
Close Editor
Auto View Off
Screen Open 0,320,210,16,Lowres
Screen Hide 0
Curs Off : Cls 0
For S=0 To 100
  Plot Rnd(320),Rnd(255),Rnd(16)
Next S
```

```
Screen Open 1,320,210,16,Lowres
Curs Off : Cls 0 : Paper 0 : Pen 6
Locate 0,0 : Print "AMOS BASIC"
Zoom 1,0,0,80,8 To 1,0,50,320,150
Cls 0,0,0 To 80,8
Screen Open 2,320,210,16,Lowres
View
Paper 0
SP=1
Do
  Screen Copy 1 To 2
  Locate 0,0 : Input "Enter the effect number (1-32000)";E
  Appear 0 To 2,E
  Wait 20
Loop
```

Try numbers such as:

1, 11, 13, 31, 41, 43, 61, 71, 73, 83, 101 and 103.

SCREEN COPY

APPEAR is certainly a useful instruction, but it's not exactly state of the art is it? We can actually generate better effects directly in AMOS Basic, using our old friend, the AMOS SCREEN COPY command.

As you'll remember from Chapter 7, SCREEN COPY allows us rapidly to copy selected areas of a screen from one place to another. With a little imagination, we can exploit this system to produce a vast range of attractive effects.

SCREEN COPY source,stx,sty,sbx,sby To destination,btx,bty

SCREEN COPY copies a rectangular region from the source to the destination screen.

stx,sty hold the coordinates of the top left corner of the area to be copied.

sbx,sby define the coordinates of the point diagonally opposite.

btx,bty choose the position on the destination screen which will be loaded with a new copy of the selected image.

Here's a small example of it in action:

```
bar 0,0 to 50,50
```

draws a white block at coordinates 0,0

```
screen copy 0,0,0,50,50 to 0,100,100
```

creates a new copy of this block at 100,100

When we used SCREEN COPY in our scrolling routines, we generally kept the size of our source area fixed, and just changed its position. But since AMOS places no limitations on the dimensions of our copying area, there's nothing stopping us from expanding or contracting this region at will. If we gradually copy the source image into the destination, we'll generate a smooth transition between the two screens

Supposing we used screen 0 as the source and screen 1 as the destination. We can now copy a larger and larger section of our source image into the destination screen like so:

```
For X=0 To 320
 Rem set up the location of the area to be copied from screen 0
 STX=0 : STY=0 : SBX=X : SBY=200
 Rem Set the top left corner of the destination to 0,0
 BTX=0 : BTY=0
 Rem Copy the area from 0,0 to X,320 into screen 1
 Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
 Rem SWAP SCREENS I'm assuming we've called DOUBLE BUFFER first
 Screen Swap : Wait Vbl
Next X
```

The effect of this routine can be seen from Figure 8.2.

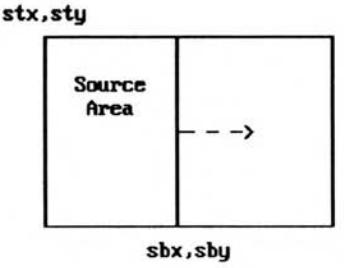

Figure 8.2: SCREEN COPY in action

Hopefully, the basic idea should seem pretty obvious. All I'm doing, is moving the edge of the source rectangle slowly to the right, and then copying this area straight onto my original source screen. Here's a demonstration of the entire system:

Example 8.5: SCREEN COPY fade effects

```
Rem Simple fade effects with screen copy
Rem Conserve the maximum amount of memory
Close Workbench :Close Editor
Hide On
Rem Hide away screen zero in the background
Screen Hide 0
Rem This is the meat of the program
Curs Off : Cls 0
Rem create a random starfield
For S=0 To 100
  Plot Rnd(320),Rnd(255),Rnd(16)
Next S
Rem open the destination screen
Screen Open 1,320,200,16,Lowres
Rem draw up the source screen
Curs Off : Cls 2 : Paper 0 : Pen 6
Locate 0,0 : Print "AMOS BASIC"
Rem create an expanded version of "AMOS BASIC"
Zoom 1,0,0,80,8 To 1,0,50,320,150
Cls 2,0,0 To 80,8
Rem Activate DOUBLE BUFFERING
Double Buffer
Rem Wait two seconds
Wait 100
Rem Set the speed of the copy. Use values from 1 to 16
Rem Higher means faster!
```

```
SPEED=1
Rem At last, the fading routine!
FRIGHT:
Rem Well it's not exactly terrifying
Rem but it does fade the screen from left to right
For X=0 To 320 Step SPEED
  STX=0 : STY=0 : SBX=X : SBY=200
  BTX=0 : BTY=0
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next X
Stop
```

We can also adapt the same principle to fade our screen from right to left. In this case, we keep the righthand edge of our area (SBX,SBY) solid as a rock, and slowly move the starting positions of our source and destination rectangles to the left.

Try replacing the FRIGHT: routine with the following lines:

```
FLEFT:
For X=320 To 0 Step -SPEED
  Rem Move the starting point of the source area to the left
  STX=X : STY=0 : SBX=320 : SBY=200
  Rem Move the starting point of the destination to the left
  BTX=X : BTY=0
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next X
Stop
```

Here's another version of our routine which copies the source to the destination from top to bottom.

```
FDOWN:
For Y=0 To 200 Step SPEED
  STX=0 : STY=0 : SBX=320 : SBY=Y
  BTX=0 : BTY=0
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next Y
Stop
```

And Bottom to top:

```
FUP:
For Y=200 To 0 Step -SPEED
  STX=0 : STY=Y : SBX=320 : SBY=200
  BTX=0 : BTY=Y
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next Y
Stop
```

We can also combine several directions at once.

```
FEXPAND:
For G=0 To 40 Step SPEED
  Rem Move the Bottom of the source area down and right
  STX=0 : STY=0 : SBX=G*8 : SBY=G*5  BTX=0 : BTY=0
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next G
Stop
```

This expands the source area downwards and sideways, so the entire screen seems to wink quickly into view.

Another idea is to start from the centre and work our way outwards. We begin by setting the source and destination areas to the middle of the screen. We then gradually expand these regions so as to copy more and more of our image into the destination. This can be seen from Figure 8.3.

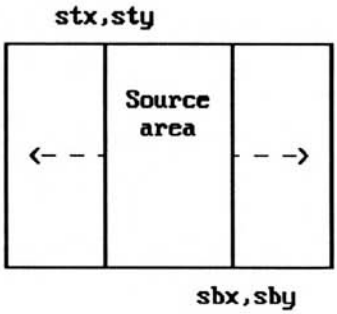

Figure 8.3: Expanding the source area

The resulting fade routine looks like this:

```
HORIZ:
Rem Expands the image sideways from the centre
For X=0 To 160 Step SPEED
  Rem moves the start of the source image to the left
  Rem and the end of the area to the right
  STX=160-X-SPEED : STY=0 : SBX=160+X+SPEED : SBY=200
  Rem Move the start of the destination area slowly to the left
  BTX=160-X-SPEED : BTY=0
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next X
Stop
```

As before, there are thousands of variations. Here are a few examples:

```
VERT:
Rem Expands the image vertically from the centre
For Y=0 To 105 Step SPEED
  STX=0 : STY=100-Y-SPEED : SBX=320 : SBY=100+Y+SPEED
  BTX=0 : BTY=100-Y-SPEED
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next Y
Stop
GROW:
Rem Expands the image upwards and outwards from the centre
For G=1 To 20 Step SPEED
  STX=160-G*8-SPEED : STY=100-G*5-SPEED : SBX=160+G*8+SPEED
  SBY=100+G*5+SPEED
  BTX=160-G*8-SPEED : BTY=100-G*5-SPEED
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next G
```

```
Stop
ODD:
Rem Expands the image in two separate steps
Rem vertical bit
For Y=0 To 100 Step SPEED
  STX=150 : STY=100-Y-SPEED : SBX=170 : SBY=100+Y+SPEED
  BTX=150 : BTY=100-Y-SPEED
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next Y
Rem horizontal bit
For X=10 To 160 Step SPEED
  STX=160-X-SPEED : STY=0 : SBX=160+X+SPEED : SBY=200
  BTX=160-X-SPEED : BTY=0
  Screen Copy 0,STX,STY,SBX,SBY To 1,BTX,BTY
  Screen Swap : Wait Vbl
Next X
stop
```

Now have a go at changing these routines and creating your own fading effects. It's fun!

8.3.4 Opening a DOOR

I'll now demonstrate how the same technique can be expanded to produce the illusion of a door opening on the screen. I'll start with an example of the system in action. Once you've seen the effect, I'll then carefully explain how it works.

Example 8.6: Door effects

```
Rem door effects
Close Workbench :Close Editor :Rem Memory conservation
Rem AUTO VIEW OFF stops AMOS from displaying our screens
Rem immediately after they've been created with SCREEN OPEN
Rem They can then be flicked into place with VIEW
Auto View Off
Rem Open source screen
Rem If you run out of memory, reduce the number of colours down
Rem to eight
Screen Open 0,320,210,16,Lowres
Rem Remove source screen from the display
Screen Hide 0
Rem Draw up a starfield
Cls 0
For S=0 To 100
  Plot Rnd(320),Rnd(255),Rnd(16)
Next S
Rem open a second source screen containing our door
Rem reduce the number of colours if you have memory problems
Rem The size is slightly taller than the standard screen
Rem Cos it looks a bit better
Screen Open 1,320,210,16,Lowres
Curs Off : Cls 0 : Paper 0 : Pen 6
Locate 0,0 : Print "AMOS BASIC"
Zoom 1,0,0,80,8 To 1,0,50,320,150
Cls 0,0,0 To 80,8
Rem Create the sides of the door
Box 0,0 To 158,209 : Box 160,0 To 319,209
Rem The next box commands are for a vertical door
rem Box 0,0 To 319,103 : Box 0,105 To 319,209
```

```
Screen Hide 1:Rem Hide away second source screen
Rem Finally open the destination screen
Rem This is the one which will be displayed on your TV
Screen Open 2,320,210,16,Lowres
Rem Copy closed door into destination
Screen Copy 1 To 2
Rem copy colour values from screen 1 into screen 2
Get Palette(1)
Rem Activate double buffering
Double Buffer
View:Rem Display our new screen
Rem The next bit just handles the sound effects
Rem You can omit it if you want
If Chip Free+Fast Free>60000 and Exist("AMOS_DATA:")
  Load "AMOS_DATA:Samples/samples.abk"
  Sam Loop On
  Sam Play 15,1,6000
Else
  Noise To 15 : Set Envel 0,0 To 100,63 : Set Envel 0,1 To 100,35
  Set Envel 0,2 To 100,30 : Set Envel 0,3 To 50,10 : Play 40,0
End If
Wait 100:Rem wait for two seconds
SPEED=1:Rem set the speed of the door opening
HDOOR:
For X=0 To 160 Step SPEED
  Screen Copy 1,X,0,160,210 To 2,0,0
  Screen Copy 1,160,0,320-X,210 To 2,160+X+SPEED,0
  Screen Copy 0,160-X-SPEED,0,160+X+SPEED,210 To 2,160-X-SPEED,0
  Screen Swap : Wait Vbl
Next X
Sam Loop Off:Rem used to turn off the sound effects
Stop
Rem Here's the vertical version of my door opener
VDOOR:
For Y=0 To 105 Step SPEED
  Screen Copy 1,0,Y,320,105 To 2,0,0
  Screen Copy 1,0,105,320,210-Y To 2,0,105+Y+SPEED
  Screen Copy 0,0,105-Y-SPEED,320,105+Y+SPEED To 2,0,105-Y-SPEED
  Screen Swap : Wait Vbl
Next Y
Sam Loop Off
stop
```

Example 8.6 may be large, but at heart, it's really surprisingly simple. The program begins by defining three separate screens for our graphics. This isn't particularly efficient, but it's very fast. I'll be showing you an alternative system later. One screen is used as the main display, whereas the other two hold copies of the door and the starfield respectively. So:

Screen 0 holds the starfield

Screen 1 holds the door

Screen 2 holds the display

The doors are opened in three stages. First, we scroll the left door from screen 1 to its new position on screen 2. This can be achieved by slowly moving the source area from the left of the screen to the centre, and copying this region to the far left of the destination.

```
Screen Copy 1,X,0,160,210 To 2,0,0
```

where X increases from 0 to 160. Figure 8.4 makes this a little clearer.

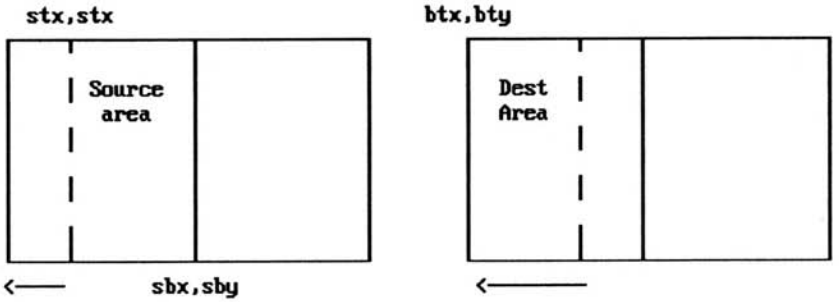

Figure 8.4: Scrolling the left door

We can now move the righthand door in a similar way:

```
Screen Copy 1,160,0,320-X,210 To 2,160+X+SPEED,0
```

This takes the area from the centre of screen one to *320-X*, and places a copy at *160+X+SPEED,0* on the destination area. As *X* increases, the source area gets smaller and smaller, and the destination point moves slowly to the right.

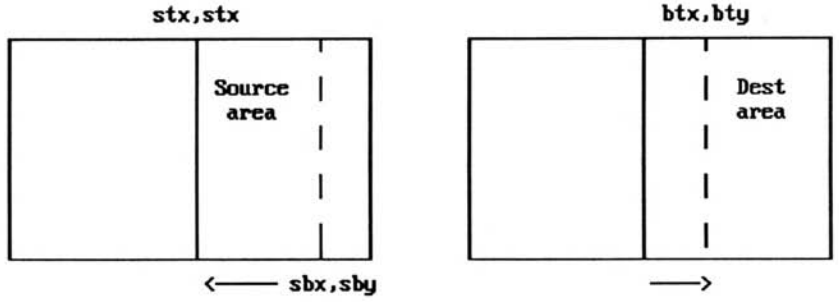

Figure 8.5: Scrolling the right door

Finally the starfield is copied from screen 0 to screen 2 using a tighter version of the HORIZ routine I showed you in the previous section.

```
Screen Copy 0,160-X-SPEED,0,160+X+SPEED,210 To 2,160-X-SPEED,0
```

Here's the completed routine:

```
HDOOR:
For X=0 To 160 Step SPEED
  Rem Scroll left side of the door to the left
  Screen Copy 1,X,0,160,210 To 2,0,0
  Rem Scroll right door
  Screen Copy 1,160,0,320-X,210 To 2,160+X+SPEED,0
  Rem Add in the starfield at the centre
  Screen Copy 0,160-X-SPEED,0,160+X+SPEED,210 To 2,160-X-SPEED,0
  Screen Swap : Wait Vbl:rem switch screens
Next X
```

The same trick can also be used to open a door vertically.

```
VDOOR:
For Y=0 To 105 Step SPEED
  Rem Scroll top half of the door upwards
  Screen Copy 1,0,Y,320,105 To 2,0,0
  Rem Scroll bottom section downwards
  Screen Copy 1,0,105,320,210-Y To 2,0,105+Y+SPEED
  Rem fill the gap in the middle with a section of the starfield
  Screen Copy 0,0,105-Y-SPEED,320,105+Y+SPEED To 2,0,105-Y-SPEED
  Rem Switch screens
  Screen Swap : Wait Vbl
Next Y
```

8.4 Conclusion

Used properly, animation can add a real feeling of excitement to even the most mundane game idea. AMOS Basic is supremely good at this type of animation, and it's senseless to let all that power go to waste. Once you've seen what can be achieved, try introducing a little animation into your own games. I'm sure you will be amazed at the results. Further examples are to be found on the companion disk to this book.

9

Sound and Music

9.1 Introduction

A few years ago, any sound effects found in a computer game would have been restricted to the occasional feeble beep. The fact is, even the best machines of the early 80s were limited to producing crude explosions or simple tones. So most programmers placed sound very low on their list of priorities, concentrating their efforts on the graphical elements of a game instead. But as computers improved, it became practicable to incorporate rather more impressive sounds into a game. However, they were limited by the programmer's ability to generate the original sound synthetically. It was therefore extremely difficult to produce authentic impressions of real sounds without complex and expensive additional hardware.

The breakthrough came with the development of techniques to encode natural sounds digitally in a form which could be subsequently reproduced by a computer. The basic idea was very simple indeed. Samples of a sound were taken thousands of times a second and converted into a list of numbers which could be held in the computer's memory. Each sample contained enough information to reproduce a fraction of a second from the recorded sound. These samples could then be successively replayed through the computer's sound chip to generate an excellent approximation of the original noise.

Unfortunately, this approach consumed enormous amounts of storage space. A couple of seconds of digitised speech could easily take up over 32k of memory. So although the technology first appeared in the heyday of the Commodore 64, it has only been since the advent of really powerful computers such as the ST and the Amiga that it's become feasible to add sampled sound to a computer game. Since its initial introduction, sampled sound has quickly penetrated most areas of the computer games industry. Sound effects are now widely recognised as a vital component of any well written computer game. So let's have a detailed look at the various techniques.

9.1.1 Getting connected

Before we start, it's worth saying a few words about the Amiga's sound system. The Amiga was originally designed to be the ultimate sound machine, and you can easily appreciate the results. Sadly though, the puny loudspeakers on most TV sets are incapable of doing it anything near justice. If we want to make the most of the Amiga's sound system, we'll need to connect it directly to a hi-fi.

If you look to the rear of your Amiga, you'll find two small sockets, just to the left of the mouse connector. These can be plugged straight into your home stereo, using a standard phono-to-phono lead available from any competent electronics shop.

That's fine if you've got a hi-fi conveniently positioned right next to your Amiga. Otherwise, I'm afraid you'll need to spend a little money and invest in a dedicated sound system. Here's a list of the various possibilities:

- Several companies now manufacture special amplified loudspeaker kits for use with the Amiga. These units will cost around £40, and will provide a reasonable approximation of hi-fi quality sound. If you've the opportunity, make sure you listen to the system in action before you buy. Otherwise you could be disappointed with the result.
- A better bet is to treat yourself to a brand new portable stereo radio-cassette player. If you shop around, you should be able to pick up a suitable system for about £60. That's only £20 more than the loudspeakers, and you'll also get a nice stereo.

You will however, need to select your system very carefully, as some of the cheaper models don't have the necessary connectors. What you need is a stereo which can playback and record sound from an external source such as a CD or turntable. In particular, look out for units with a CD/line switch and built-in phono sockets at the back. This is a good indication that the stereo will be suitable for use with your Amiga. Just to be on the safe side though, you'll need to check painstakingly with the retailer before committing yourself. Choosing a system may take a while, but when you get it right, the sound effects will shake your room!

9.2 Sampled Sound

Sampled sound provides even the most inexperienced programmer with the ability to generate sound effects which would make a IBM PC user gasp in amazement. Even if they've plugged in an expensive sound card!

9.2.1 Basic ideas

Playing a sample from AMOS Basic is easy. Generally, all our samples are stored in a special memory bank (usually 5). This bank can now be installed permanently in memory, and saved automatically as part of our programs. Samples can be created using any cartridge on the market and converted into AMOS format using the SAMPLE_BANK_MAKER.AMOS utility on the AMOS Program disk. We can also make use of the hundreds of sample files on the public domain. Once they've been loaded into memory, our samples can be played directly using a simple SAMPLAY command:

SAMPLAY s

Plays sample number *s* from the current sample bank.

Let's listen to a few examples from the AMOS Data disk. So switch on your sound system, fire up direct mode, and type the following lines:

```
load "AMOS_DATA:samples/frog_samples.abk
samplay 1
```

Boing!

```
samplay 2
```

Ribbit!

```
samplay 3
```

This really makes a splash!

samplay 4

Duck!

samplay 5

Plays a simple instrument. I think it's a guitar.

samplay 6

A pan-pipe?

Now for a small demo program:

```
On Error Goto SERROR
Curs Off : Cls 0 : Paper 0
F$=Fsel$("*.abk","","Load some samples")
If F$="" Then Stop
Load F$
Centre "<Sample experimenter>"
Locate 0,2 : Centre F$
Do
  Locate 11,10 : Cline : Input "Sample number ";S
  Sam Play S
Loop
SERROR:
Locate 7,20 : Print "Sample ";S;" does not exist"
Wait 50
Locate 0,20 : Cline
Resume Next
```

Warning! Don't attempt to load the files Boom.ABK or Aaaah.abk. These are in a RAW format and can be used with the SAMRAW command I'll be showing you later.

9.2.2 Looping the loop

One obvious problem with these samples are that they are all extremely short. Supposing we wanted to generate an alarm which continued for several seconds? Well, we could call a series of SAMPLAY commands in a FOR..NEXT loop.

```
Curs Off : Cls 0 : Paper 0
Centre "<Diverse Alarms>"
Load "AMOS_DATA:Samples/samples.abk"
Repeat
  Locate 12,10 : Cline : Input "Sample number ";S
  If S>0 and S<13 Then For E=0 To 10 : Sam Play S : Wait 10 : Next E
Until S=0
```

This would force our program to work flat out playing our alarm. So there wouldn't be any time to deal with the rest of our game. As you'd expect from AMOS, there's a solution at hand.

SAM LOOP ON

SAM LOOP OFF

SAM LOOP ON tells AMOS to repeat any subsequent SAMPLAY command continually. Our program can now carry on as normal, with the sample playing automatically in the background. When we are finished with our effect we can turn it off using SAM LOOP OFF.

Example:

```
Curs Off : Cls 0 : Paper 0
Centre "<Diverse Alarms>"
Load "AMOS_DATA:Samples/samples.abk"
Sam Loop On
Repeat
  Locate 12,10 : Cline : Input "Sample number ";S
  If S>0 and S<13 Then Sam Play S
Until S=0
Sam Loop Off
```

Notice how the timing has improved. Each sample is played out completely before repeating.

9.2.3 Changing the playback speed

Another useful trick is to change the speed at which a sample is played back through the loudspeaker. This can have a dramatic effect on the final sound. AMOS provides a separate version of the SAMPLAY command for just this purpose:

SAMPLAY v,s,f

v is a bit-map containing a list of voices which will be used to play the sample. The Amiga's hardware allows us to play up to four separate sounds simultaneously. These are combined to form the eventual effect we hear through our TV or loudspeaker. Each voice is assigned a number from 0 to 3, and can be turned on or off by setting the appropriate bit in *v*.

If that sounds horribly complex, then I've good news, as we won't be needing it for our samples at all. Although this technique is vitally important for our music, it's actually rather useless in the case of samples. Since AMOS only allows us to play a single sample at a time, the only significant effect of changing the voices is a slight reduction in the overall volume. Boring!

We'll therefore concentrate on varying the speed of our samples instead. It's much more fun! From now on, simply use a value of 15 for the voice number and all will be well. This will force AMOS to play our sample through each of the available voices and ensure the maximum possible sound quality. Ok? Anyway, back to the interesting bit.

f sets the playback speed of the sample in units of samples-per-second (sps). Each sample can be played at any speed we like, from 1000 sps (slow) to 30000 or more (mega fast!). The faster the speed, the higher the pitch of the resulting sound.

It's important to recognise that the sound produced by the SAMPLAY instruction varies enormously depending on the speed. So the same sample can be exploited to generate dozens of different effects.

Examples:

```
Load "AMOS_DATA:samples/samples.abk"
samplay 1
```

Plays sample 1 at its normal speed.

```
samplay 15,1,1000
```

Can you hear the difference?

```
samplay 15,1,1500
```

Same sample, higher pitch.

```
samplay 15,1,2000
samplay 15,1,3000
samplay 15,1,4000
samplay 15,1,7000
samplay 15,1,15000
```

Remember, all this is from just ONE sample!

Here's an experimenter program which will allow you to play with this system some more. At the start of the program, you'll be asked to enter a sample file from the disk, using a standard AMOS file selector. You'll then be able to play each sample through a range of speeds by tapping one of the keys Q to / on the keyboard. The current sample number can be changed at any time by pressing the appropriate digit.

```
On Error Goto SERROR
Curs Off : Cls 0 : Paper 0
F$=Fsel$("*.abk","","Load some samples")
If F$="" Then Stop
Load F$
S=1
ESC=69
Centre "<Sample experimenter>"
Locate 0,5 : Centre F$
Locate 11,10 : Print "Sample number ";S
Do
  X$=Inkey$
  If X$<>""
    If(X$>="0") and(X$<="9")
      S=Asc(X$)-Asc("0")
      If S=0 : S=10 : End If
      Locate 11,10 : Print "Sample number ";S;" ";
    Else
      N=Scancode
      If N=ESC : Erase 5 : Stop : End If
      If N>=16 and N<28 : N=N-16 : End If
      If N>=32 and N<=42 : N=N-20 : End If
      If N>=46 and N<=58 : N=N-26 : End If
      If N<33
        Sam Play 15,S,1000+N*500 : Locate 15,13 : Print "Note ";N;" "
        Locate 13,16 : Print "Speed ";1000+N*500;" ";
      End If
    End If
  End If
Loop
SERROR:
If ERN=0
  S=1
  Locate 11,10 : Print "Sample number ";S;" ";
  Resume Next
Else
  Error ERR
End If
```

9.2.4 Choosing a sampler cartridge

If you want to record your own samples, you will need to buy a piece of special hardware known as a sampler cartridge. This will cost you around £20-£50, and will usually plug into your Amiga's printer port. All the currently available packages contain software which allows them to be used in conjunction with the majority of programming systems, including AMOS.

When you're starting out, it's sensible to get the cheapest cartridge you can lay your hands on. In practice, the quality of the effects is almost entirely dictated by the Amiga's hardware. There's therefore very little difference between the most expensive cartridges and the cheapest. Here are a few things to look out for.

- **Stereo versus mono.** AMOS only allows us to play back mono samples, so there's no point in splashing out on an expensive stereo sampler.
- **Software support.** The best cartridges also incorporate powerful sound editors which enable us to manipulate and edit the sample sounds inside in the computer. Useful options include GATE, LIMIT, FILTER and ECHO.
- Note that most sampler cartridges measure the speed in kilo-hertz (sounds painful!). A speed of one kilo-hertz (Khz) means that 1,000 samples will be played through the Amiga's sound system in each second. So 1 Khz=1000 sps.

Once you've created a sample, you can load it into AMOS format using the SAMPLE_BANK_MAKER utility on the AMOS Program disk.

Normally the sample maker only allocates a mere 40K of space for our sample data. If you've a megabyte or more of memory, you can increase this to expand the amount of storage space for your samples. You simply change the number in the SET BUFFER command at the start of the program. The original line reads:

```
Set Buffer 40
```

If you wanted to allocate 100K, you'd therefore replace this line with:

```
Set Buffer 100
```

You can now save the edited version of the program straight onto your working copy of AMOS Basic.

Using the SAMPLE MAKER is easy. Just click on the right mouse button to bring up the menu line, and call up the LOAD SAMPLE option from the EDIT menu. This will present you with a standard AMOS file-selector which can be used to load your samples into memory. You'll then prompted to enter the speed of your sample in kilohertz. This will be saved in the sample bank and used as a default. If you're unsure, choose a speed of 10. This should give reasonable results with most samples.

You can then repeat this procedure with each of the samples you wish to install into the sample bank. A read-out of the current samples will be displayed at the top of the screen. As you are going along, jot down the relevant sample numbers on a scrap of paper. This will serve a useful reminder once you return to AMOS Basic.

When you've finished, select SAVE SAMPLE BANK from the EDIT menu, and enter the filename of your new sample bank on the disk. The Sample Maker will now set to work, converting your samples, and will save the data straight to your new file.

Finally, you can exit from the utility with QUIT and return back to AMOS Basic. Your new file can now be loaded directly into AMOS and played using the SAMPLAY instruction as normal.

STOP..PRESS..

Look out for the new Soundex sample editor coming soon from the AMOS User Group. This will allow us to edit our samples directly from AMOS Basic, and manipulate them with a wide range of professional fading effects. Contact Aaron Fothergill for details:

1 Lower Moor, Whiddon Valley, Barnstaple, North Devon EX32 8NW

9.2.5 Making the most of your sampler

The only reliable way of selecting the correct recording speed in any particular case, is by careful experiment. The following guidelines may prove helpful.

- Record special effects such as explosions and ray-guns using a sample speed of between 5 and 12 kilohertz. The quality of these effects is often perfectly acceptable at even the lowest speeds.
- Speech requires a minimum sampling rate of around 8KHz to be understandable. In practice, the desirable speed will be closer to 10 or 12. This was the speed used to generate the samples provided with AMOS Basic.
- Make a careful note of the recording speed for each sample. This will be needed when you convert your sample into an .abk file for use with AMOS Basic.
- Only use recording speeds which are a whole number KHz. Sample speeds such as 7.5KHz can be played back from AMOS with ease. But you won't be able to install the correct sample speed into the sample bank using the Sample Bank Maker utility.
- Keep a watchful eye on the memory consumption. Each individual sample uses up a single character (byte) of the Amiga's memory. The total length of a sample can therefore be calculated from the following simple formula: LENGTH=SPEED*TIME

If,for instance, you had created a sample 10 seconds long using a speed setting of 5KHz, the total sample size would be: 5000*10 bytes or 50k

Alternatively, if you recorded the same sample at 20KHz, it would now occupy an astonishing 200K of memory. It should be obvious from the above calculation that the slower you can keep the sample speed, the more samples you will be able to pack into your programs. There's also the question of the accuracy of the recording.

The apparent quality of a sound increases in direct proportion to the rate at which it was sampled. This is because the higher speeds generate a much more detailed representation of the original sound. The slowest samples do sound rather choppy and incoherent by comparison to the fastest. There is therefore a trade-off between the quality of the sample, and the amount memory it uses.

The key to the successful application of sampled sound is to manage this trade-off in your own favour. You have to find a way of keeping the sample speed down to an absolute minimum, while still providing enough information to enable the sampler extensions to reproduce the sound realistically.

9.2.6 Playing a RAW sample

RAW samples can be loaded directly into AMOS Basic from any external source such as a sound sampler. But they need to be played back using the complex looking SAMRAW instruction. The format is:

```
SAMRAW voice,address,length,speed
```

Yuck! In practice we can usually get away with a line like

```
samraw 15,start(5),length(5),10000
```

where 5 is the number of the memory bank containing our sample, and 10000 is the playback speed in samples-per-second.

Examples:

```
load "AMOS_DATA:samples/Aaaah.abk"
```

```
samraw 15,start(5),length(5),10000
```

Plays the current sample in bank five at a speed of 10,000 samples per second. Aaaah!

```
samraw 15,start(5),length(5),5000
```

Moo! See how I've changed the effect by reducing the speed to 5,000 sps. Take my advice. Don't use this command unless you've no alternative. It's fiddly to use, and slightly buggy to boot!

9.2.7 Potential sound sources

I'll now discuss a number of potential sources for our samples.

The Public domain

As sound samplers have become increasingly popular, a number of disks have appeared on the public domain. Since the AMOS package is compatible with the majority of current sampling systems, it's usually possible to incorporate these samples directly into our Basic programs.

Films and tapes

If you have access to a sampler cartridge, you may be tempted to *borrow* effects from your favourite records or video tapes. While you are unlikely to encounter any problems with samples used in games you've written for your own amusement, you will almost certainly need to get permission from the original producers if you wish to distribute your game commercially.

The situation regarding special effects like explosions and Phasers is rather more complex. This is because after we have manipulated one of these sounds, it can prove almost impossible to recognise the original source. It's rather like trying to reconstruct an egg once we have made it into an omelette! Messy!

At the present time, the precise legalities of this situation have yet to be fully resolved. So it's advisable to make a note of all your original sources in the program documentation before submitting any work to a commercial software house. After all, if you are lucky enough to get your game accepted, the sound effects will probably need to be re-mixed anyway.

Television

If your TV set has an earphone socket, you may be able to connect a microphone and take your samples directly from the airwaves. If all else fails, you can always record the sounds straight from the loudspeaker.

Although some of the sounds will be subject to copyright, there will be plenty of others which can be used immediately in your games without any problems. These include a vast range of real sounds from an aeroplane landing at Heathrow, to the thundering of a Formula II motor car. Providing you use a little imagination, you can capture a variety of sounds which would otherwise be impossible, all from the comfort of your own living room.

Don't forget about the radio. Nowadays the airwaves are full of interesting and unusual noises, ready and waiting to be grabbed into your games. Just turn the radio on to Medium or Long wave, and set your recorder running. You can now twiddle the dial randomly and sample the resulting sound effects. Easy eh?

Other sources

It's a well known fact that the best things in life are free. The world around us contains a vast spectrum of potentially useful sounds, from bird calls to car engines. By carefully using the

sampler program, we can transform even the most mundane sound, like a door bell or a dog bark, to something strange and possibly even sinister. In the earliest days of radio, it was commonplace for the various sound effects to be generated by hand during transmission. These clever people were capable of generating marvellous results from just couple of coconut shells or a few small bits of wood. Even the original effects from the amazing *Hitch-hiker's Guide to the Galaxy* were created with these techniques. The radio series was recorded using equipment which would now appear laughably primitive compared to the facilities provided by a modern recording studio. And yes, in case you hadn't guessed from the original AMOS manual, I'm a fan. (Check out the references to 42 and the ANSWER if you're unconvinced.)

Anyway, with a little ingenuity and a lot of experimentation, it's possible to produce almost any sound imaginable. Once we've entered this into the computer, we can then sculpt it into something incredible! Don't underestimate the power of your own voice. The range of sounds which can be directly produced by the human vocal cords is truly remarkable! Almost as good as the Amiga, in fact!

9.2.8 Creating an alien

Many games on the Amiga now incorporate interactive alien speech effects. If you've listened to some of these sounds in admiration, you will be delighted to hear that it's quite possible to create a convincing impression of alien speech using a standard sampler cartridge.

Begin by carefully setting up your cassette recorder and preparing for a recording session. Now start the recording and try to talk in gibberish for several minutes. (I found this especially easy!) If you attempt this feat with several people, you'll probably fall into hysterics at some point. This is perfectly natural, and nothing to be ashamed of. Even the silliest noises can be taken down and used in AMOS!

After you've made your recording, you will hopefully be left with a tape containing some extremely funny noises. Connect your tape recorder to the sampler cartridge and create a sample in the normal way using a speed setting of 16KHz.

You can now gradually increase the play back speed up to the maximum of 32 KHz, noting the results. The original noise will quickly change into something quite alien. Reverse the sound using speeds from 16KHz down to 5KHz. Again make a note of any effects you find pleasing. You can then choose the results you particularly like, and save the samples to the disk. This allows you to manipulate your sounds to your heart's content, without any fear of losing anything permanent if you make a mistake.

You are now ready to enter the effects menu of your sampler software. In my experience, I've found that the best results are obtained with ECHO, REVERB and HALL. It is important to note that the effects of these commands add together. So if you repeatedly add reverberations to one of your samples, the quality of the sound will be progressively changed.

Hopefully, you will eventually end up with an interesting set of alien sound effects for use with your game. These can now be saved to disk and replayed at the appropriate points in your program.

9.2.9 Possible applications

I'll now discuss a number possible ways sampled sounds can be exploited in our AMOS games.

Arcade games

In my experience, adding realistic sound effects such as explosions to an arcade game can produce a massive improvement in the game's overall quality.

Simulations

The use of sampled sound in a simulation will obviously vary depending on the nature of the game. If we are simulating a real situation, it's often possible to generate the samples directly from the source. Take a game like golf, for instance. In this case, we can capture the thwack of a golf club as it hits the ball with just a simple visit to your municipal golf course with our tape recorder.

RPGs

One of the most impressive uses of sampled sound that I've so far encountered, was contained in FTL's excellent Dungeon Master. This included a host of realistic sounds for everything from a door opening to the shrieks of the monsters. Many of these sounds were of extremely short duration, and could be sampled at quite low speeds without any appreciable loss of quality. Here are a couple of ideas for creating the various effects.

- The swish and the thud of the blades can be produced using some of the samples supplied with the AMOS package.
- The screeches can be sampled directly from your own voice. You may need to modify the sounds a little with REVERB or ECHO to get the desired effect.

Adventures

Adventures are traditionally extremely memory hungry. So although graphics have now become commonplace, sampled sound is still in its infancy.

The scope for sampled sound in an adventure game is enormous. By adding the appropriate effects as the adventure progresses, we can enhance considerably the illusion of reality. Supposing we are playing an adventure set in a dungeon (just for a change). As you progress through the caverns, you encounter a number of doors. Imagine how you would feel when you listened at a door and heard a horrible scream! Even the most mundane sounds, such as the strike of a match or a telephone ringing, can have a dramatic effect if the player isn't expecting them.

In order to conserve memory, it's advisable to store samples in separate files on the disk. When required, they can be entered into memory with LOAD and played using the SAMPLAY command. If we record our samples at 10KHz or less, we can pack up to 15 samples on a single disk, while still leaving enough room for another 10 compacted pictures. This means that we don't necessarily have to sacrifice graphics to make the space for our sound effects.

9.3 Music

If you've been using your Amiga for some time, you'll already be aware of the stunning soundtracks it can generate. My personal favourite in this respect is Xenon II which contains an excellent soundtrack by Bomb the Bass! Well, I like it anyway!

If it's used properly, music can enhance a game beyond recognition. So you'll probably be wondering how you can create some for your own AMOS games.

As always, we'll need to go into some theory first! Although the Amiga's quite capable of generating entire soundtracks synthetically, most music still relies heavily on sampled sounds. But as I mentioned at the start of this chapter, samples requires massive amounts of memory. Let's perform a few simple calculations. A typical soundtrack lasts one to three minutes before repeating.

The amount of memory consumed can be worked out from the following formula:
MEMORY=SAMPLES PER SECOND * TIME

So if we sampled a single minute's work of music at 16KHz, we'd need :

16,000*60 = 16,000*60 =960,000 bytes

Ouch!

How can we possibility cram all this data into our Amiga? Interestingly enough, it's a problem we've encountered before. Remember our difficulties with storing the hundreds of level screens in our scrolling games? Same problem, similar solution.

Just to concentrate your mind a little, load up AMOS Basic, and place the AMOS Data disk in the internal drive. Now type the following program to play some music:

```
Curs Off : Cls 0 : Paper 0
Load "AMOS_DATA:music/amosteroids_title.abk"
Music 1
Locate 0,10 : Centre "Listen and weep!"
Do
  Locate 14,14 : Print "Seconds ";S
  Wait 50
  Inc S
Loop
```

This music repeats after around four minutes. If we attempted to store it in its entirety, we'd require about 3 megabytes of data. Yet it plays on even the smallest Amiga! Any ideas? With a bit of luck, you'll have spotted the trick. Instead of being held in full, the sound is divided into a number of separate themes which are repeated several times in the final music. These are known as patterns. On their own, they'd reduce the memory consumption by up to four or five times.

We can however do much more. Each pattern is actually made up from a sequence of individual notes played on several different instruments. With a bit of Amiga magic, each instrument can be held in memory as a single sample and replayed at different speeds to simulate the various notes in our music.

Using this technique, we can store all our music as a compact list of numbers. The only samples we require are the ones for our instruments. These instruments are therefore the equivalent of the tiles in our screen maps. The same set of instruments can be exploited to generate thousands of different soundtracks.

The resulting compression rates have to be seen to be believed. So let's have a look. Break out of the music program and jump to direct mode. If you're getting sick off the tune, enter MUSIC OFF first. Now type the following line:

```
Dir "amos_data:music/"
```

This will bring up a list of all the music files on the AMOS Data disk. See Amosteroids_Title.abk. That's the one we've been playing. It's around 30K long and its only a 100th of the size of the equivalent raw sample. Wow! Anyway, so much for the theory. Why don't we toodle off to the Amiga music system, and see how these soundtracks are created and played.

9.3.1 Playback

Playing a piece of music in AMOS is simplicity itself. AMOS stores all our music in memory bank 3 along with any samples used by our instruments. These instruments are completely separate from the normal AMOS samples, so they don't clash with our explosions or speech effects.

Each bank can contain several complete soundtracks, although it's common practice to limit ourselves to just one per bank. This allows us to load each tune as and when we need it and saves the maximum memory for our programs and screens.

We can load our music into AMOS Basic using the LOAD command like this:

```
load "music.abk"
```

where music.abk could obviously be any filename we like.

Once we've loaded our music into memory, we can play it though our Amiga using a simple MUSIC command from AMOS Basic.

MUSIC n

Plays a piece of music from memory bank 3. The music number *n* ranges from one to the number of available tunes in the bank. Since there's usually just a single soundtrack, the standard use of the command is:

```
music 1
```

The MUSIC instruction is really rather clever. All music is played automatically in the background, and like AMAL, it doesn't interfere with the running of our programs in the slightest. What's more, it senses the use of any other AMOS sound commands such as BOOM or SAMPLAY, and temporarily halts the music for the duration of the noise. We can even play another piece of music through the system as well. The old tune will continue from its original point when the new one ends.

Example:

```
dir$="AMOS_DATA:Music/"
```

Sets the current disk directory to a folder on the AMOS Data disk containing some music.

```
load "funkey.abk"
```

Loads a bit of music into memory bank 3.

```
music 1
```

Plays our music.

```
boom
```

Sets off an explosion through the Amiga's sound system. After the explosion's finished, the music restarts immediately from the precise point it left off.

```
load "music.abk":music 1
```

More music, more excitement

```
music off
```

Turn it off, turn it off!

The MUSIC OFF command just stops the music in its tracks. Any previous music command will be restarted automatically. While we're playing our music, we can muck around with the speed and volume to our heart's content. The speed can be set using the TEMPO command like so:

TEMPO s

where *s* is a speed value from 1 (very slow) to 100 (blindingly fast). Here are some examples:

```
dir$="AMOS_DATA:Music/"
load "funkey.abk"
Rem load our music
wait key
Tempo 50:rem faster faster
wait key
Tempo 4:rem funeral march
wait key
Tempo 16: rem back to normal
```

MVOLUME n

Adjusts the volume of our music. *n* ranges from 0 (silent) to 63 (quite loud). The best effects can be generated by keeping the volume at its maximum and controlling the sound level directly from our TV or hi-fi.

All AMOS music is split into four completely separate tracks played simultaneously through our hi-fi or TV set. Each track contains just a part of the eventual sound. So there's one track for the beat effects, another for the main theme and so on. Think of the Amiga in terms of a rock band. The tracks correspond to the musicians, who can each play a different melody on a range of different instruments. The final sound is made up of the sum of their relevant contributions.

The tracks are played through one of four voices numbered from 0 to 3. These voices provide the instruments for our invisible musicians. We can select which voices will be played using the AMOS VOICE instruction.

VOICE mask

mask is a number from 0 to 15. The status of each voice in the music is held in a single binary bit. If this bit is set to 1, the voice will be played, but if its set to zero, the relevant voice will be disabled. So we won't be able to hear the track assigned to the current voice.

Don't worry if you're not into binary, as there's a simple trick which will allow you to choose the voices directly. Just place a % (percent) sign after the VOICE command and enter the 1s and the 0s in the following order: 3210. Remember, a 1 turns the voice ON and a 0 switches it OFF. Here are some examples:

```
          3210
voice %1111
```

Activates all four voices in the music.

```
          3210
voice %1010
```

Turns on voices 3 and 1 and stops voices 2 and 0.

```
voice %0011
```

Disables voices 2 and 3 and starts up voices 0 and 1.

Now for a demonstration of the AMOS Music system in action. Here's an experimenter program which allows you to play around with all the various options. Try loading

Amosteroids_Title.abk. You can now press the numbers 0 to 3 to turn each track on and off as required. As you can hear, each track lends its own individual effect to the music.

```
Rem Open A HELP screen
Screen Open 1,320,200,2,Lowres
Rem The help screen contains a list of instructions for the program
Rem Press the HELP button on the keyboard to bring it on the screen
Curs Off : Colour 1,$FF0
Cmove 1,0 : Centre Border$("Instructions",1)
Print : Print : Print : Print : Print
Cmove 1,0 : Centre Border$(" Changing the tempo",1) : Print : Print
Centre "Left Arrow reduces " : Print
Centre "Right Arrow increases" : Print
Print : Print : Print
Cmove 1,0 : Centre Border$(" Volume controls ",1)
Print : Print
Centre "Up  Arrow reduces " : Print
Centre "Down Arrow increases"
Print : Print : Print
Cmove 1,0 : Centre Border$("Track selection ",1)
Print : Print
Centre "Use the numbers 0 to 3" : Print
Centre "To toggle each track ON or OFF" : Print : Print : Print
Centre "<Hit a key to continue>"
Wait Key
Rem Hide HELP screen
Screen To Back 1
Rem Display screen 0
Screen 0
View
Rem This is the scancode of the HELP key
HELP=95
Curs Off : Cls 0 : Paper 0
Rem Load a music *.abk file off the disc
F$=Fsel$("*.abk","","Load some music")
If F$="" Then Stop
Load F$
Rem Set initial values of TEMPO and VOLUME
T=20 : V=63
Rem kill the mouse
Hide On
Centre "<Music experimenter>"
Rem Dimension an array to hold the status of each track (ON) or OFF
Dim TS$(4)
Rem Dimension an array to hold the VOICE settings for each track
Dim TK(4)
Ink 1,0
Rem load arrays, and display status
For S=0 To 3
  TK(S)=2^S
  TS$(S)="on "
  Text 100+S*30,90,Str$(S)
  Text 100+S*30+5,100,TS$(AV)
Next S
Locate 0,4 : Centre F$
Locate 15,6 : Print "Tempo ";T
Locate 15,7 : Print "Volume ";V;" ";
Rem Draw VUMETER
Box 99,109 To 219,195
For L=1 To 3
```

```
  Draw 99+30*L,109 To 99+30*L,185
Next L
Text 134,195,"Tracks"
Music 1
L$="off"
Rem Main loop
Do
  X$=Inkey$
  If X$<>""
    Rem Toggle track on or off
    If(X$>="0") and(X$<"4")
      AV=Asc(X$)-Asc("0")
      Gosub SET_TRACK
    End If
    Rem Toggle a filter on or off, and change the power LED
    If(X$="L") or(X$="l")
      If L$="on"
        Led Off
        L$="off"
      Else
        Led On
        L$="on"
      End If
    End If
    Rem Increase playback speed of music
    If X$=Cright$
      Add T,1,0 To 100
      Tempo T
      Locate 15,6 : Print "Tempo ";T;" ";
    End If
    Rem Slow down the tempo
    If X$=Cleft$
      Add T,-1,0 To 100
      Tempo T
      Locate 15,6 : Print "Tempo ";T;" ";
    End If
    Rem Pump up the volume
    If X$=Cup$
      Add V,1,0 To 63
      Mvolume V
      Locate 15,7 : Print "Volume ";V;" ";
    End If
    Rem Slide down the volume
    If X$=Cdown$
      Add V,-1,0 To 63
      Mvolume V
      Locate 15,7 : Print "Volume ";V;" ";
    End If
  End If
  Rem Read HELP key
  If Scancode=HELP
    Screen To Front 1 : Wait Key : Screen 0 : Screen To Front 0
  End If
  Rem Display a simple VU meter
  Gosub VU
Loop
VU:
For TRACK=0 To 3
  Rem Get volume of each track
  IT=Vumeter(TRACK)
```

```
  Rem and move the bar up or down
  If IT>0 and TK(TRACK)>0
    Ink 0 : Bar TRACK*30+105,110 To TRACK*30+123,110+IT
    Ink 4 : Bar TRACK*30+105,184-IT To TRACK*30+123,184
  Else
    Ink 0 : Bar TRACK*30+105,110 To TRACK*30+123,184
  End If
  Wait Vbl
Next TRACK
Return
Rem Set each track ON or OFF
SET_TRACK:
If TS$(AV)="on "
  TS$(AV)="off"
  TK(AV)=0
Else
  TS$(AV)="on "
  Rem This sets the appropriate bit in the voice number
  TK(AV)=2^AV
End If
BM=TK(0)+TK(1)+TK(2)+TK(3)
Rem VOICE controls which tracks are being played
Voice BM
Rem Print ON or OFF above the track display
Ink 1,0 : Text 100+AV*30+5,100,TS$(AV)
Return
```

Note that the flashing bars were created with the help of the AMOS VUMETER function.

`volume=VUMETER(v)`

This returns a value from 0 to 63 which represents the volume of the current sound being played through voice *v*. Most of the demo tunes on the AMOS Data disk stick with a volume of about 63. So the effect is pretty boring. But other soundtracks change the volume continually during the course of the music.

9.3.2 Converting music from an external music editor

Although AMOS doesn't come with its own built music editor (sigh!), it can load musical compositions from practically any music package in the Amiga. Before we can play these tunes however, we first need to convert them into a special AMOS format using one of three special programs on the AMOS Extras disk. Once we've transformed our music into a standard .abk file, we'll be able to play it directly in one of our games. Here's a list of the available conversion utilities:

`Sonix.AMOS`
Changes a piece of music created with Sonix into AMOS format. At the time of writing, there are still a few bugs in this system, and the program usually fails.

`Game Music Creator (GMC)`
Converts some music from GMC to AMOS. Note: This must be saved using the SAVE DATA icon. Otherwise the converter will fail!

`Soundtracker/Noisetracker/Protracker/etc`
Transforms a Soundtracker Module into an AMOS .abk file. Don't attempt to convert a Song using this system. It won't work, and the program will crash with an error!

If you're starting out, most of this will mean nothing, as you won't have a music editor to convert from. Fortunately, there's a solution at hand. Amazingly enough, the Soundtracker, Noisetracker and Protracker programs are now available in the public domain. So you can treat yourself to an immediate copy from your favourite PD library for around £3!

While you're at it, don't forget to pick up a few example disks as well. These will provide you with some invaluable pointers to creating high-quality music on the Amiga. Another useful source of these files is the AMOS PD Library.

9.3.3 Music editors

Ok, so you've got hold of the latest Soundtracker clone, and loaded it into your Amiga. And you're confused! Well, I'm not surprised! All the PD music editors have been designed by and for hackers, who are dedicated to getting the absolute maximum out of the Amiga's sound system. Inevitably, this adds a great deal of complication to the resulting programs. I'm not complaining of course! After all, we're not actually paying for their software, are we? Without the sterling efforts of these people, the Amiga music scene would be a pale shadow of what it is today.

Opinions differ as to the best of Music Editors, but for me, Protracker is hard to beat. Not only is it extremely powerful, but the programmers have thoughtfully included a built-in help system which takes some of the sting out of the learning process! What's more, Protracker was given away free with a recent issue of CU Amiga (Disk 14). So you may already have it! Figure 9.1 contains a simplified picture of a typical program screen.

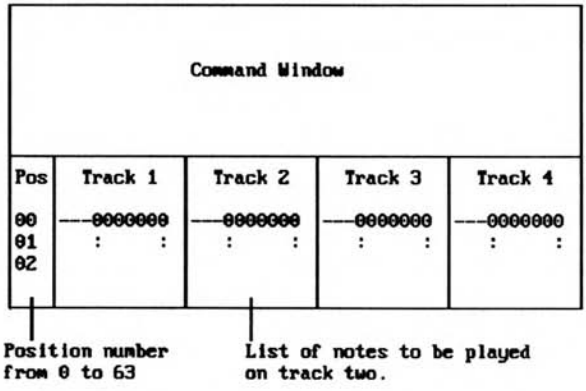

Figure 9.1: A typical music program

Along the top, there's a command window containing dozens of fancy icons. These can be selected by clicking on them with the left mouse button. The lower half of the screen holds the piece of music we are creating. This is divided into four separate tracks which can hold up to 64 notes each. The notes are stored in a format such as:

```
NOTE SAMPLE COMMAND
```

For example:

```
G#2  01  000000
```

Plays note G# in octave 2 using sample 1.

Each music system has its own individual ways of entering these notes into the computer. Thankfully, most editors allow us to play the Amiga's keyboard as if it were a real instrument. So we don't need to know anything about musical notation. We can simply tap away and go! I'll now explain some of the command icons you'll encounter during your experiments.

General commands:

EDIT
Allows us to begin entering our notes into the present pattern.

PLAY
Plays the entire song in memory. The current note is highlighted by a horizontal bar across the middle of the screen. For the purposes of discussion, a song is the complete piece of music we are composing. It's built up out of up to 127 slots, each of which can contain its own individual pattern number. So the same pattern can be reused several times in the music.

STOP
Stops the music!

RECORD
Starts the music playing and allows us to enter our notes directly over the existing tune.

CLEAR
Erases all or part of the current tune.

USE PRESET
Tells the editor to use a set of built-in samples for the various instruments. These can be assigned to specific sample numbers using the SAMPLE icons. See later.

CHOOSE PATTERN
Again, each music Editor provides its own different method of changing the existing pattern number. But Protracker indicates the pattern we are currently editing using two digits just to the left of the sample name. These can be selected with the mouse and a new pattern number entered straight from the keyboard.

Disk options

DISC OP
This forms the gateway to a separate control panel which accesses a list of common disk operations.

SAVE SONG
Saves the current song along with all the pattern definitions. These songs can be loaded into our music editor directly. But they can't be entered into AMOS Basic, as they don't include any of the relevant samples.

LOAD SONG
Loads a song from the current disk. If the new song uses samples which are not already available in memory, they will be loaded automatically from the appropriate disk files.

SAVE MODULE/SAVE DATA
Saves the song data along with all the relevant instrument samples. These files can be subsequently loaded into one of the AMOS converter programs, and transformed into a standard AMOS music file.

LOAD MODULE
Loads a complete module into the editor's memory.

LOAD INSTRUMENT/LOAD INS/LOAD SAMPLE
Loads the sample used by the present instrument. Soundtracker expects these instruments to be on specially arranged PRESET or ST- disks. But Protracker allows us to use any practically samples we like!

Warning! Protracker doesn't bother to increase the sample number after we've loaded a new sample. So if we are grabbing several samples at a time, we'll need to adjust the sample number directly using the SAMPLE icon.

SAVE SAMPLE
This is only available from the more advanced systems like Protracker, and allow us to use the music editor to control an external sampler cartridge. As you'd expect, it saves our sample data straight to the relevant disk file.

Sequence commands

These are used to enter the numbers of the current slot, pattern and sample. Each number can be changed at any time by simply clicking on one of the arrow icons to the right of the button. Usually, all values are stored in hexadecimal notation. This works in units of 16 rather than 10, and uses digits from 0 to 9 and letters from A to F:

```
Hex digit    0 1 2 3 4 5 6 7 8 9  A  B  C  D  E  F
Decimal      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Once you've used the system for a while, you'll quickly pick up the idea.

POSITION 0000 (or POS)
This is used in conjunction with the PATTERN arrows, to assign an individual pattern number to each available slot. The arrow icons select the number of the slot we are currently defining.

PATTERN 0000
Chooses the number of the pattern assigned to the present slot. Each pattern is assigned a number from 0 to 63, and can be reused again and again in the final music sequence. The arrow icons to the right allow us to move through all the available pattern numbers. Just to make life easier for us, there's often a second icon with the same name, which allows us to PLAY the pattern through our hi-fi.

LENGTH 0000
Sets the number of slots in the song we are currently editing. This should be increased whenever we create a new slot with the PATTERN icon. It's not incremented automatically.

SAMPLE 0000
Picks the sample number which will be used as our instrument. There's usually an equivalent keyboard alternative which lets us change the sample as we are entering our notes.

VOLUME 0000
Sets the volume of the sample, from 0 to $40 (64)

9.3.4 Using a music editor

I'll now briefly go through the entire process of making a piece of music.

- ❑ Load up some samples for your instruments. If you're using one of the better systems such Protracker, these can be anything you like, and can be created using any standard sampler cartridge. Check out your favourite PD library for a list of compatible instrument disks for your particular editor.

Samples can be selected in two ways: You can either enter the DISK OP menu and load them individually, or you can assign them using the PRESET button and activate them with USE PRESET. Either way, don't forget to increase the sample number by 1 every time you assign a sample. Otherwise, all your samples will be loaded into the same sample number! So you'll only have a copy of the last one you've entered.

- ❑ Select a sample to be used as the first instrument. Use the arrow icons to the right of the SAMPLE button.

- ❑ Now click on EDIT to start entering your notes. The current position will be highlighted by a thick horizontal bar. There's also a traditional cursor bar which indicates the precise note you are changing. This can be moved around with the left and right cursor arrows as normal. Additionally, the Tab key (to the left of the Q) can be used to move the editing position between the four possible tracks.

- ❑ Move the cursor over to the left of the screen and tap in your notes from the keyboard. These will be immediately played through the loudspeaker, so you'll be able to hear your tune as you are entering it into memory. If you're uncertain, hit a few keys at random. You'll quickly learn which keys generate which notes.

As you type, the bar will advance progressively downwards, allowing you to enter each note in the music sequence. You can move this bar over any note in the track using the up and down arrow keys from the keyboard. The current instrument can be changed directly either from the SAMPLE icon or using the cursor pad (Protracker). For the moment though, concentrate on the first track. Try to type in a simple tune, with a pleasant uncomplicated melody. If you make a mistake, you can easily delete your notes using the standard Del key.

- ❑ Now play your tune by selecting the PLAY icon. If the music needs further work, click on the EDIT icon and repeat the previous step. If it's ok, you can save your music on the disk, and move straight to the next track.

- ❑ It's now time to create some background effects. Select an instrument such as a drum, and hit the keys in a repetitive sequence to generate a natural beat. If you're feeling adventurous, you can enter the notes while the existing track is playing using RECORD.

- ❑ Generate each of the remaining tracks in exactly the same way. Don't forget to save your music to disk at regular intervals with the SAVE SONG option. Most of the PD Music editors aren't especially reliable, so it pays to be a little paranoid. It's also a good idea to save your music in a new file before you make any major changes. This will avoid the risk of making a disastrous and permanent mistakes.

- ❑ When you're happy with the current pattern, you can assign it to one or more slots by selecting the arrows near the POSITION and PATTERN icons. You'll also need to increase the length of the song as appropriate.

- ❑ Now click on the pattern number below the sample name, and enter 01 to move to the next pattern. You can then create a new pattern which will form the basis of the next section of your music. Remember that you can reuse each pattern as many times as you like. So you only have to define a particular melody once in your music.

- ❑ Finally, save the entire arrangement as a Module, and fire up AMOS Basic. You can now load the appropriate converter program and save the music as an .abk file. This can be entered straight into your AMOS program and played with the MUSIC command.

Whew! It's pretty involved isn't it. I won't pretend that adding a soundtrack to your games is child's-play, because it isn't. If you want to generate professional results, you'll need to work long and hard on the keyboard. But don't lose heart! If you persevere at it, there's nothing stopping you from generating terrific musical accompaniments to your games.

9.4 Conclusion

Hopefully, you'll now be buzzing with ideas for using sound and music in your AMOS programs. Sound effects won't make a bad computer game a success. Nothing can do that, other than a total rewrite. But it will add a final gloss to an already enjoyable game. This might make all the difference if you try to sell it commercially. So add a little sound to your programs, and make your players jump off their seats!

Appendix 1

Easy AMOS

A1.1 What is Easy AMOS?

Although the Game Maker's Manual is intended for all AMOS users, it does assume a general knowledge of Basic programming. If you're an absolute beginner, you may have found some of the previous sections pretty heavy going! Fortunately, there's a solution in hand.

Easy AMOS is an exciting new tutorial package from Mandarin/Europress software, designed especially for total novices. It comes complete with a powerful, but cut-down version of AMOS Basic and provides everything you need to start your exciting journey through the world of games creation.

In order to keep things simple however, Easy AMOS does not include advanced features such as AMAL, Sprites or Menus. So unless you've already got a full copy of AMOS Basic, you may encounter real problems in applying some of the techniques in this book. Since the Game Maker's Manual was written specifically for the professional version of AMOS Basic, it naturally exploits a number of techniques which are just not available from the smaller Easy AMOS system.

There's no need to panic however! The general principles of games creation apply equally well to all versions of the AMOS language, and you'll still be able to create useful and exciting games with the Easy AMOS package. What's more, when you finally come to upgrade to the full version, this book will be invaluable, as it contains a detailed, step-by-step guide to the complete AMOS system. Easy AMOS has already taught you how to write simple Basic programs. The Game Maker's Manual will gently show you to how to write full-blown games in AMOS Basic!

On the minus side though, the differences between AMOS and Easy AMOS make it impossible to guarantee than any particular example program will work in Easy AMOS. If there's sufficient demand, I'll be happy to produce a special Easy AMOS companion disk along with this book. This will contain modified versions of all the relevant example programs, along with an on-screen tutorial explaining any new techniques which will be needed to produce games in Easy AMOS. Contact Sigma for more details.

A1.2 Significant Differences

For the moment, watch out for the following areas:

Sprites

These aren't used in Easy AMOS, so you can completely ignore any references to them in this book. Actually, I've only used them in a couple of examples. So you're unlikely to suffer any real problems.

AMAL

Alas, this is one of my favourite features of AMOS Basic, and I've used it a LOT! As far as I'm concerned, it's well worth buying the full AMOS package, just to get access to this amazing system! Having said that, Basic alternatives do exist for all the AMAL routines in my programs. But until you upgrade, you'll need to take Section 3.6 (Arcade games) and Section 8.1 (Animation techniques) with a pinch of salt. You'll also need to make modifications to the missile firing routines in Section 3.5. Concentrate instead on Movement tables (3.7) and Animation tables (8.1.6)

Bob Clear, Bob Draw

The Bob command has had quite a few modifications in Easy AMOS. So you may have problems adding animation to rapidly scrolling backgrounds.

Utilities

Currently, the contents of the Easy AMOS tutorial disk have yet to be finalised. So you may find real differences between the utilities I've mentioned in this book, and their Easy AMOS equivalents. Check the Easy AMOS documentation for more information.

A1.3 Final Words

As Easy AMOS is released on to the market, it's quite likely that further incompatibilities will come to light. If you get really frustrated, drop me a line care of Sigma, and I'll try to help you out. Any useful suggestions will be included in the Easy AMOS companion disk as and when it becomes available.

Appendix 2

Sell..Sell...Sell

A2.1 And Finally...

Well, that's about it for this book. I hope you've enjoyed it. It's now time to get down to some serious programming and create your very own game in AMOS Basic. Have fun! But wait! Haven't I forgotten something? After spending months of effort on your new game, your mind may well turn to thoughts of impending riches. Don't worry. This is perfectly natural, and there's absolutely nothing to be ashamed of!

I can't alas, guarantee an immediate fortune for you. Computer gamers are notoriously fickle lot, and history is littered with potentially great programs which simply failed to sell. However, if you've produced something you're really proud of, you would be stupid to keep it just to yourself. After all, what have you got to lose? At worst, you can always place your program in the public domain. You may not earn any money, but it will certainly get your name on the map. Plenty of PD programmers have gone on to write successful commercial games. So don't be discouraged if you fail at the first attempt. Your next program could be a real winner.

A2.2 Direct Selling

The simplest approach is to sell your game directly to the general public via mail order. This definitely won't make you a millionaire, but it could go a long way towards that hard disk or memory upgrade you've been lusting after.

There is however a snag! In order to sell your game in any significant quantities, you'll need to sink a fair amount of your own money to cover advertising and packaging costs. Suffice it say, it would be VERY unwise to use your life savings for this purpose! So treat the whole venture as a gamble, and only spend money which you can genuinely afford to LOSE!

A2.2.1 Starting up

You begin by carefully working out how much each game is going to cost to produce. You can now add in your profit margin to reach an estimate of the final purchase price. Here's an example:

Item	Cost
One 3.5 inch disk	0.65
Photocopied documentation	0.20
Packaging	0.30
Postage	0.50
Total Production Cost	1.65

The reason the costs are so low of course, is you have none of the overheads which would be incurred by a commercial software house. Once all the expenses have been deducted the profit margin on a commercial £20 game is surprising small.

It's essential to be realistic when you're deciding on your final profit margin. Ideally, your game should retail at between £5 to £7. This will allow you to make a handsome £3 to £5 on every program you sell. If you get too greedy, and attempt charge more, you are unlikely to make money. That's because you will now be competing directly with the likes of Xenon II and Space Harrier. Remember that commercial games often have a very short shelf life. So after a few years on the market, they can usually be picked up at a fraction of the original price.

If however, you place your game firmly into the budget category, you could do extremely well. Everyone likes a bargain, and there's a tremendous demand for inexpensive but well written games. This field is much easier to break into, and provides an excellent introduction to the commercial games scene for the beginner.

A2.2.2 Creating a master disk

The next step is to generate a master disk which will be used to create all future copies of your game. This should contain the various programs and data files which will be required to load your game into the computer. The master disk should be specially configured so that it automatically loads and runs your game during the start up process. This can be achieved in two ways:

RAMOS
RAMOS is a special run-only version of AMOS Basic that can be freely distributed along with your own games. In order to use this feature, you'll first need to create a RAMOS boot disk. This can be accomplished using the instructions from the original AMOS manual. You can now save your game under the name AUTOEXEC.AMOS and copy over any additional data files on to your disks. If you've installed RAMOS correctly, your game will run automatically whenever you boot your Amiga from the new disk.

The AMOS Compiler
This is an amazing new utility which allows you effortlessly to convert any AMOS program into super-fast machine code. If you've purchased this package, you'll be able to create your disks in a matter of minutes. Just compile your program in the normal way, and copy it straight on to a freshly formatted disk. Now follow the simple instructions in the AMOS Compiler manual to generate a bootable program disk. Warning! All CLI or Workbench format programs require the full complement of standard Amiga library files on the disk. If these files are not available, the program's just won't work!

The easiest way of getting the correct files on the disk, is to make a COPY of the AMOS Update disk, and delete everything EXCEPT the L:, DEVS:, LIBS:, S: and C: folders. You can now compile your program on to this disk in CLI or Workbench format, and install it using the AUTOBOOT.AMOS utility.

As a default, the compiler automatically creates a standard screen to hold your programs and graphics. Generally however, you'll want to start the system off completely clean.

This can be accomplished by changing a couple of simple options from the compiler. Select the Spanner icon on the control panel, and click on the following buttons:

❑ Create default screen: Was YES, should be NO

❑ Send AMOS TO BACK upon booting: Was NO, set it to YES

You'll now need to create a new screen at the start of your program using SCREEN OPEN. When your compiled program is first initialised, this screen will be totally invisible to the user. It can be displayed by simply including a SCREEN TO FRONT command at the appropriate point in the program. Your game screen will then flick neatly into view. **Note:** This technique ONLY works with programs which have been compiled in CLI or Workbench format. It doesn't effect .AMOS programs at all!

Anyway, after you've produced your master disk, don't forget to make a couple of extra copies for back-up purposes. You should also run it through a virus checker as well, just to be on the safe side!

It's now time to test the game disk on as wide a range of machines as possible. Be a little paranoid about this, as even the smallest mistake could be extremely irritating. Believe me, anything you miss will be gleefully reported back to you by dozens of irate users! In particular, you'll need to ensure that your game will run on Amigas with more than a single drive. Each extra drive takes up over 17k of valuable memory. So a program which runs fine on an unexpanded machine will often fail miserably when the user plugs in an extra drive. Additionally, if your program requires a megabyte of memory in order to work, it's vital to make this totally clear on all your packaging and documentation. If there's a problem with the user's machine, you can then refund their money immediately, without wasting valuable time searching for nonexistent bugs.

A2.2.3 Disk duplication

Most commercial software houses generate thousands of copies of their programs at a time, often using the services of a professional disk duplicator. This process is extremely expensive, and is obviously way beyond the reach of even the most enthusiastic amateur.

Since you're only dealing with relatively small numbers of disks, it's feasible to copy each game by hand when you receive the order. If you plan things carefully, the copying process shouldn't take more than around five minutes per disk. However, you will definitely need access to at least two drives, as otherwise you could waste hours of backbreaking work simply swapping between the various disks. **Hint:** Don't forget to slide the write protect tab on your master disk before you begin. This will automatically safeguard your program disk from any accidental errors during copying.

If you're feeling really professional, you might wish to invest in a disk-copier utility. These can be purchased for between £30 to £60 pounds, and can easily duplicate an entire disk within a couple of minutes. Nowadays, disk-copiers have an awful reputation because they can also be used for making illegal copies of commercial games. This is clearly wrong, and you should obviously only use them to copy your own material. If you get the urge to cheat, try to imagine how you would feel if someone were to pirate your own game. Hopefully, the true morality of the situation will now spring clearly into focus, and you'll find the temptation very easy to resist!

Before sending out each copy of your game, it's important to give it a quick test on your machine, just to make sure it actually works. You should also run it through a virus checker as well. This will avoid a great deal of unnecessary frustration on the part of your users, and will save you both time and money in handling returns.

A2.2.4 Documentation

If your program's small, you may be able to get away with a single sheet of instructions. This should include a concise list of the various game controls. If more information is required, it's cheapest to display it directly inside your program. With a little effort, you can create interactive tutorials which demonstrate all the various features of your game using the actual screens. Used properly, these can add a whole new dimension to your program.

Another idea is to place a detailed explanation of the game in a separate READ.ME file on the disk. This can be printed on the screen using a PD utility such as More, and will be thoroughly appreciated by all serious users.

The documentation should now be thoroughly checked for errors, preferably with the help of a friend or relative. Try to avoid any obvious spelling mistakes in your documentation, as these tend to detract from the quality of your game. So if, like me, your spelling is pretty appalling, take the trouble to check any dubious words through a dictionary or spell checker.

Once you've created your documentation, you can easily duplicate it using a standard photocopier. If you don't have access to one these machines, take your work to a commercial stationer or copy shop. These are reasonably inexpensive, and will often offer special services such as double sided printing, which would be unavailable from your local library. If your documentation is larger than about 10 pages, you'll probably need the services of a professional printer. This won't be cheap, and should be avoided unless it's absolutely necessary. Be warned! Prices do vary substantially from company to company, so it's vital to shop around.

A2.2.5 Packaging

In order to save on costs, all packaging should be kept to an absolute minimum. It's pointless to supply your program with expensive disk boxes or inserts, as these will simply not be appreciated by your users. Instead, your game can be posted directly as it stands in a padded envelope or jiffy bag. These cost about 20p each, and provide good protection against the rigours of postage.

If you want to add a really a professional touch, you might like to invest in some fancy disk labels. These can be obtained at reasonable expense from any commercial printer. Alternatively, if you've access to a decent matrix printer, you may decide to generate them yourself. Labels can be purchased on special sheets which slot straight into your printer's tractor feed. You can then print them directly from AMOS Basic using an appropriate series of Basic LPRINT commands. You can also make use of one of the many label printer utilities available from the public domain.

A2.2.6 Advertising

Advertising is obviously an essential part of any mail order operation. It's also extremely expensive. A typical quarter page advert in a popular computer magazine could cost you anything up to £300. Unless you are incredibly lucky however, this money will be completely wasted!

Supposing you were to make a net profit of just £3 per game. If your advert cost £300, you'd need to sell over 100 copies just to break even! In practice, this figure is ludicrously optimistic. There's no cast iron guarantee that you'll get any orders at all! So although computer magazines are great fun to read, their advertising rates are far beyond your reach. AVOID!

It's much better to place your advert in one of the many PD magazines and newsletters dedicated to the Amiga. These publications will often allow you to insert a pretty substantial

advert for around £50. What's more, PD magazines will often rake in several times the sales of the more commercial publications. That's because you are now targeting your adverts directly to the people who are really interested in budget software. If you price your game keenly, it won't cost a great deal more than the average PD disk. So it will be an attractive alternative to one of the more popular PD games.

Another way of generating publicity is to send off review copies to your favourite computer magazines. Don't forget to mention that your game was written in AMOS Basic, as many of the Amiga columnists are enthusiastic AMOS supporters. This will substantially increase the chances of your game being reviewed. The nicest thing about these reviews is that you've got absolutely nothing to lose. A favourable comment in a reputable magazine can sell more copies than a dozen advertisements. But if the reviewer hates your game, it will usually go straight into the rubbish bin, without a word of complaint!

A2.3 Licenseware

One of the most profitable ways of distributing a budget game, is using the Licenseware approach. Instead of selling the program direct, you simply make a special arrangement with one or more software libraries to sell your game as a Licenseware product. This allows them to make as many copies of your game as they like, and market them to the general public. The only restriction, is that they have to pay you a small royalty (typically £1) for every copy they sell.

On average, Licenseware costs the user around £5. So it's only slightly more expensive than a standard PD program. Since the software library handles all the duplication and packaging though, you can distribute your game with absolutely no overheads. And if you're successful, you could easily rake in hundreds of pounds from the deal!

A good starting point, is to try the AMOS PD Library run by Sandra Sharkey. The address is in Chapter 8 (Section 8.2.5). Obviously it's entirely up to Sandra to decide whether your program is suitable for wider distribution. But if the game is REALLY good, you're certainly in with a chance.

A2.4 Software Houses

Every programmer has the dream of selling a game to a commercial software house. Unfortunately, the competition is incredibly fierce. Most software houses now receive dozens of unsolicited submissions every week, and yet they can only release a maximum of five or six games per year.

On the face of it, the odds against your game being accepted seem incredibly daunting. But if you think your game is of a commercial quality, or has original ideas which deserve a wider audience, it's well worth persevering. Be warned though! Few people achieve success on the very first attempt. So if your game is rejected, don't give up in disgust. Go back to the drawing board and try to work out why it failed. If you are able to learn from your mistakes, your next program could do much better.

If the worst comes to the worst, you can always try submitting your game to a magazine. With the advent of integral cover disks, there's a lot of demand for cheap, high quality games. You may not receive more than a couple of hundred pounds for your work, but you'll have the considerable satisfaction of knowing that it's been played by up to 40,000 Amiga users. This could pay you real dividends when you attempt to sell your next game. As you would expect, there's no sure-fire way of guaranteeing a sale. If I had the answer, I'd be in the Bahamas somewhere, living off my royalties! In the final analysis, the success and failure of your game

will depend on its quality. That's up to you. You can however, improve your chances considerably by following a few simple guidelines:

A2.4.1 Choosing a software house

Most software houses will expect you to submit your game on a strictly exclusive basis. So if they discover that you've sent the same program to all their competitors, their interest will quickly evaporate. However, if the first company rejects your game, there's absolutely nothing stopping you from submitting your program elsewhere.

Because of this fact, it's essential to target the software house very carefully. Only consider those who have a proven track record in your chosen game category. If you attempt to sell an arcade game to an adventure specialist, it will almost certainly will be rejected out of hand.

You should also consider the AMOS element. Even now, some companies have an irrational prejudice against anything written in Basic. This is a hold-over from the early 80s when Basic was regarded with genuine derision, especially by assembly language programmers. With the advent of AMOS Basic, this has naturally changed, but old prejudices die surprisingly hard.

If you've bought the AMOS compiler, there's a solution at hand. Just compile your program using the clean-start technique I showed you in section A2.2.2, and create an autoboot disk using the AUTOBOOT.AMOS utility. Your game will now load and run automatically when it's booted from the internal drive. And providing you avoid a few obvious give-aways, its AMOS origins will be completely hidden!

Tips

❑ Don't use the standard AMOS file-selector!

If you need to select some files, you can create your own file-selector, with the AMOS ZONE commands. A directory listing can be loaded into a string array using a combination of the DIR FIRST$ and the DIR NEXT$ instructions.

❑ Replace the mouse pointer with one of your own.

Use the CHANGE MOUSE command to change the appearance of the default mouse pointer. It can be taken from any 16x16 four colour image from the sprite bank. Here's an example:

```
Screen Open 0,320,255,4,Lowres
Hide : Curs Off : Cls 0
Circle 8,8,7
Get Sprite 1,0,0 To 16,16
For C=0 To 3 : Colour 16+C,Colour(C) : Next C
Change Mouse 4
Show
```

❑ Mandarin/Europress Software do NOT require you to include the AMOS copyright messages as part of your program. So you don't actually have to tell the software house that the game was written in AMOS Basic! If your game is accepted, you MUST write to Mandarin and provide them with a full copy of the finished game.

❑ If the game sells, Chris Payne of Mandarin will gleefully tell the world, two months after the release date. He's the marketing genius who launched AMOS Basic in the first place! So the ensuing publicity for your game is likely to be amazing! It could even lead to more sales. Of course, the software house may get a bit of a shock! But they're hardly likely to complain if they've made a handsome profit! On the other hand, if the game is a complete wash-out, Chris will probably keep quiet, and they'll never know!

A2.4.2 Presentation

When your game arrives on the desk of your chosen software house, it will be dumped straight onto the in-tray of the software development manager (or equivalent). These people have the thankless task of ensuring that all the company's projects reach completion under budget, and before the final deadlines. Since this is patently impossible, software development manager's tend to work extremely long hours, and live under continual and unrelenting pressure. Inevitably, this leaves them very little time to check out the unsolicited submissions from people such as yourself. So anything you can do to make things easier will be much appreciated.

If your game is badly presented, or difficult to load, it will probably be rejected out of hand. Unless it makes an instant impression on the reviewer, it just won't get a second glance. Although most companies are always on the look out for new talent, I'm afraid there simply isn't enough time for a detailed examination of every piece of software which hits the reviewer's desk. You therefore have to ensure that the merits of your program are immediately and blindingly obvious.

A2.4.3 Program documentation

All submissions should be accompanied by a brief cover note. This should be as concise as possible, and should provide a basic explanation of your game, along with a quick list of instructions. Ideally your program should load and run automatically, but if there are any special loading requirements, make sure that they are totally clear. As with all documentation, it's essential to check your work thoroughly for typographical errors. Nobody is likely to take you seriously if your letter is littered with obvious spelling mistakes. Alongside the cover note, you can also include a range of extras:

Biographical material
This is a handout (*curriculum vitae* - CV) containing your personal details and highlighting any relevant programming experience. It can include practically any information you like. It's not necessary to provide a full list of your examination resultsmention, but do mention which programming languages you're familiar with, and the type of computer systems you've used. On the lighter side, you can also include a list of your hobbies or a reference to your favourite computer games. If you just happen to like some of the company's existing products, all the better!

Game plan
You can also provide a detailed breakdown of the basic mechanics of your game, along with a full explanation of the gameplay. Normally, you will be able to create this document by simply tiding up your original game plan. Realistically, this material is unlikely to be read in full unless your game is accepted. It will however, undoubtedly impress the reviewer on the amount of work you have put into the project. One serious drawback however, is that you are effectively revealing all your trade secrets. This might allow an unscrupulous company to use your ideas in their own programs, without paying you a penny. Since the majority of software houses are reasonably honest, this is improbable, but it's certainly something to watch out for.

A2.4.4 Disk documentation

It's fact of life that by the time your disk reaches the hands of a reviewer, it may well have been separated completely from your documentation. You should therefore label each disk extremely carefully. All disks should include the following information:

❑ The name of your program.

- ❑ The computer system required to run it.
- ❑ An indication of the amount of memory it needs.
- ❑ Your full name and address.
- ❑ The date the program was submitted.
- ❑ The disk number (if there is more than one.)

See Figure A2.1.

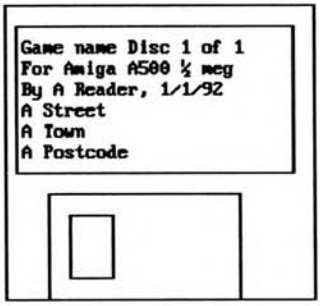

Figure A2.1:

In addition to the visible documentation, you should also place a full list of instructions on the disk in a standard READ.ME file. If you've created your documentation on a wordprocessor, you can also include copies of your cover note or CV. Make sure that all text is in ASCII format. This will allow the instructions to be displayed directly on the screen, without requiring a copy of the wordprocessor. (See your wordprocessor manual for more information.)

A2.4.5 Sending your submissions

Once you've generated your documentation and prepared the packaging, you are ready to send your game off to your chosen software house. You can get the address from the advertising section of your favourite magazine.

You now need to exercise a great deal of patience. If you're successful, you can expect to receive word from the software company within a couple of months. NEVER attempt to hurry the process along by phoning or writing to the software house. This will simply waste their time, and is extremely unlikely to increase your chances of acceptance.

If you haven't heard from the company after about three months, I'm afraid your game has almost certainly been rejected. Unfortunately, few companies have the resources to write up detailed rejection slips, even if you include a stamped addressed envelope. You must therefore take a good, hard look at your game before proceeding any further. Is it the quality really high enough, or are you just deluding yourself? If it still seems promising, it may worth sending it somewhere else. You may just have submitted your game to the wrong software house. Alternatively, you may decide to send your game to a magazine. This is certainly something to consider as it releases your game to the widest possible audience.

A2.4.6 Terms and conditions

Finally, I'll say a few words about the financial arrangements you might be offered if you are lucky enough to get your program accepted. There are several possibilities:

Royalties

The standard method of payment is to take a 10 percent royalty on each copy of your program which is sold. This is always calculated from the retail price of your game. So if your game sells in the shops for £10 you'll get £1 for each copy. That may not sound much, but it can quickly mount up. There are after all, over 500,000 Amiga owners in the UK alone. If you can reach just a small fraction of these people, you will be showered with loot!

Normally, the terms and conditions are laid out in a formal contract which specifies the royalty rate, and assigns exclusive publication rights to the chosen software house. Like most contracts, it's worth reading the small print very carefully before signing on the dotted line. Get legal advice if you're unsure.

It's especially important to make sure that you keep control of the copyright of the game. This means the software house is only allowed to publish your game for the duration of the contract. The program itself remains your property, so if the original software company goes bust, you can freely offer it to somebody else.

Advances
Royalties are usually paid at six monthly intervals commencing from the date of publication. In practice, this means that you could be waiting up to a year for your first pay-cheque. Many companies ease the pain by lending you a percentage of your royalties in advance. These advances are entirely at the discretion of the software company, and are specified directly in the contract.

Outright payment
Sometimes, a software house will offer to buy your game outright. If you accept, you will receive a single payment on publication, and that's it. The advantage of this type of deal is that there's no risk. Once you've made a sale, you can spend your money and effectively forget about it. You will however, lose money if your game does phenomenally well. All the extra profit will go straight to the software house. So if you've got a choice, you are recommended to stick with the royalty system. It may be riskier, but it's potentially much more lucrative.

Loan machines
If the software house requires you to perform a little extra work on your game before publication, you may be offered the use of some additional hardware. This obviously needs to be returned after the game has been finished, but it's certainly a useful bonus. Typical hardware ranges from a hard disk (common) to a complete Amiga A3000 (very rare!).

A2.5 End Game

Wow! You've reached the end of the book. AMOS Basic has an exciting future ahead. With a little talent, and a lot of effort, you could easily be part of it. Go ahead. Make my day!

Appendix 3

Glossary of Terms

Adventure
A game which allows the player to explore a world of mystery and adventure.

AMAL
Amos Animation Language. This is a powerful built-in animation system consisting of dozens of simple commands. AMAL allows us to rapidly move anything from a screen to a spaceship! All commands are performed independently of our AMOS programs, 50 times a second!

AMOS
An amazing programming language written by Francois Lionette. It was developed from STOS Basic on the ST. So the letters don't actually STAND for anything! The name was derived from the equation STOS-ST+AMiga = AMOS. That's it!

Animation
The art of creating the illusion of movement on a static TV screen. AMAL supports a range of commands to animate our objects, including ANIM.

Animation table
A table of image numbers which will be flicked through a sprite or Bob in sequence.

Arcade game
A game involving fast furious action, which pits the player against an impressive and varied collection of invaders.

ASCII
A numeric code from 0 to 255 used to store each character inside our computer.

Attributes (Role-playing)
Provide a role playing game with a measure of the strengths and weaknesses of the various characters. Typical attributes are STRENGTH, SPEED and STAMINA

Bank
A memory area used to contain vital information such as the images used by the Bobs, icons used by our games, any music and sound effects, and our TOME map definitions. AMOS memory banks are numbered from 1 to 15, and are saved automatically along with our Basic programs.

BASIC
This is supposed to stand for *Beginners All-Purpose Symbolic Instruction Code.* But if you believe that, you'll believe anything! It's basically just a standard series of instructions used to communicate with a computer in something approaching real English.

Blitter object (Bob)
These are special screen objects which can be moved around the screen at high-speed using the Amiga's infamous Blitter chip. They are perfect for the objects in an arcade game. See *Sprites.*

Buffer
Informal computer jargon for any large memory area.

Bug
A computer term for an error in one of our programs.

Byte
The smallest units of information which can be stored in a computer. Each individual byte can hold either a single character, or a number ranging between 0 or 255.

Coarse scroller
Moves a map across the display in units of a single tile element. It may be lousy for arcade games, but it's fine for simple RPGs.

Code
Either a symbol or system used to represent vital information inside the computer, or a general term for any small section of a program.

Colour registers/indexes
The Commodore Amiga allows us to display up to 32 independent colours on the screen at a time. The precise shade of each colour is held in a set of 32 memory locations known as the colour registers.

Colour scrolling
Colour scrolling is a simple technique which progressively changes the values held in the Amiga's colour registers to produce a vast range of interesting animation effects.

Compiler
This is a separate utility which converts any AMOS program into the equivalent machine instructions. These instructions can be executed directly on the Amiga, without the need to use the AMOS Basic package.

Complex scroller (Arcade game)
This is a type of arcade game set in a large two dimensional playing area which can be scrolled smoothly in any direction.

Coordinates
These provide a simple way of specifying the position of an object on the screen. The X coordinate measures the horizontal distance from the top left corner of the display, and the Y coordinate sets the vertical separation. Traditionally, all coordinates are entered in pairs. e.g. X,Y.

Data structure
A list of connected information treated as a unit. Typical data structures include arrays and records.

Debug
To fix up a computer program so that it actually works! The process of correcting the errors in one of our games.

Digitiser
An external piece of hardware which allows us to read an image straight from a video recording.

Documentation
A complete written description of the workings of a computer game. Often aided by the use of REM statements within the program.

Double buffering
A clever system which performs all drawing operations in an invisible background area. It's activated by the DOUBLE BUFFER command, and is often used in conjunction with Blitter objects.

Fine scroller
A powerful scrolling technique which scrolls a large view screen through the display area. Although this consumes vast amounts of memory, the results can be amazing.

Fumble (Role playing)
A random number generated by the combat calculations of an RPG which represents a disastrous mistake on the part of the player's character. Zero on a 10 sided dice.

Game map (Adventures)
A complete map of the locations of each room, and the connections between them, usually drawn up on a piece of paper when we are designing our adventure. It often includes brief descriptions of the various puzzles and objects to be encountered by the players.

Game plan
A complete plan of our game written in advance. By planning our games carefully, we can simplify the eventual programming enormously.

Game time
The time which is presumed to have passed inside the game world. Sometimes this can be very different from the actual time experienced by the player.

Game world
The imaginary world created inside a computer game such as a simulation or an adventure.

Garbage collection
An automatic reorganisation which releases any of the Amiga's memory which is no longer needed by our AMOS Basic programs.

Graphic adventure
Just like normal adventures, except that they read the player's commands using fancy control panels, and display the locations in the form of pictures.

Graphic text
This is text which can be positioned anywhere on the screen using standard graphic coordinates. It's created using the AMOS TEXT command.

Hardware coordinates
These are the special coordinates used for the sprites and the screen display commands. Their starting point is far outside the TV screen. But the visible region starts from about (128,40).

Hardware scrolling
See *offset scrolling*.

Horizontal Scroller (Arcade Game)
An arcade game set against a large horizontal landscape. As the player moves forwards, the scenery scrolls smoothly underneath.

Icon
Icons are special images held in memory bank 2. They are used to generate either the buttons needed in a control panel, or the tiles in a map display.

IFF
A standard file format used to stores screens, samples or music. Only the Screen version is supported by AMOS Basic. It's used with the LOAD IFF command to enter our images from any drawing package on the Amiga.

Implementation
This is a technical term for the process of converting a written specification (or game plan) into a program which can be entered into our Amiga.

Interrupt
Internal routines which are called automatically after a certain period of time. Used by AMOS Basic to control AMAL and the sound effects.

Level (Role playing)
A single number which indicates the general level of abilities attained by a player's character. The level of a character usually increases with experience.

Logic error
An error in the design of our program.

Logic/Physic Scrolling
This is a complex, but powerful scrolling system which allows us to perform all our scrolling operations on just one double buffered screen. It's best for vertical scrollers.

Logical screen
The screen used by the Amiga for drawing operations. If different from the physical screen, the image under construction will not be displayed until our program is ready. AMOS Basic allows us to refer to this screen directly using the LOGIC function. See *Screen Swapping*.

Look-up table
A table of values calculated in advance that allows us to replace complex calculations with a simple access to an array, and can speed up many programs considerably.

Map
A description of the game display held in the computer. RPGs and scrolling games draw their maps using a list of icon values stored in memory. These values are known as tiles.

Map Editor
This is a special program used to create large screen maps out of an arrangement of icons. A simple map editor can be found on the AMOS program disk (MAP_EDITOR.AMOS). There's also an expanded version called TOME available from Shadow Software.

Melee rounds (Role playing games)
Split a combat situation into a number of phases which can be performed by a computer.

Mock-up
Simple sketches of the game screens produced during the planning process. They can be easily created with Deluxe Paint as test screens for the earliest versions of our game.

Movement table (Arcade games)
Contain a list of the successive positions of an object during an attack wave. They are used extensively in games such as Space Invaders

NVP (parser)
A text recognition system capable of interpreting sentences including two nouns, a verb, and a pronoun. e.g:

```
slide card through slot
  :     :       :     :
Verb  Noun pronoun noun
```

Offset scrolling
A scrolling system which moves the visible area of a large screen through the display, one section at a time.

Parry (RPGs)
An attempt by characters to defend themselves from attack.

Parser (Adventures)
The section of an adventure program which reads the user's commands and translates them into a form that can be interpreted by a computer.

Physical screen
The screen currently being displayed on your TV. It is not necessarily the same as the screen used for the various drawing operations. It can be entered in our drawing commands using a special PHYSIC function.

Pixels
Short for *picture elements*, these are the smallest individual points which can be manipulated on the Amiga's screen.

Playfield
A name for the playing area used in our games. This can be much larger than the visible screen. Using a scroller it can be moved through the display, a piece at a time.

Procedure
A completely independent program module which can include its own variables and DATA statements. Procedures are created with the help of the Procedure and Endproc commands.

Property list (Adventures)
The list of information needed to determine the position and type of an object in an adventure.

Prototyping
The process of testing possible ideas by experiment before including them in the final game plan.

Pseudo-code
A simplified form of English which allows us to describe concisely the action of a program without having to involve ourselves with any messy implementation details.

Rainbow
An attractive banding effect created with the SET RAINBOW and RAIN commands. The shade of a colour changes according to its position on the screen.

Real-time
The time which is actually experienced by the players of a simulation. Separate from the time assumed to have passed in the Game world.

Real RPGs
The original Role playing game played out on paper using dice.

Routine
A small, but independent section of a program, which can be reused again and again in our games. Routines can be created using either the Procedure system, or with our old favourite GOSUB.

RPG
Short for role playing game

Sample
A snatch of a real sound captured as a list of numbers inside the computer.

Scanner
This is a separate bit of hardware which lets us grab an image straight off a drawing or a photograph.

Screen swapping
The art of updating a piece of graphics on an invisible logical screen and smoothly updating the display.

Scenario
The plot of an adventure game or RPG.

Scrolling
Refers to the process of moving a massive playing area through the screen one step at a time. It allows us to set our games against vast imaginary landscapes, and is used extensively in Arcade games or RPGs.

Simulation
A mathematical model of a real world situation held inside a computer.

SIN and COS
Two mathematical functions which describe the ratios of the various sides of a right angled triangle. Used in many 3D graphics programs and for creating fancy movement patterns.

Sound Tracker
A PD music program which can be used to add attractive sound effects to our games.

Specification
A detailed plan of a computer program made out before its creation. Similar to a game plan.

Sprite
A special graphical element which can be manipulated independently from the rest of the screen.

Static shoot-em-up (Arcade game)
This an arcade game where all the action takes place on a static background screen which remains fixed for the duration of the program. Variety is added by introducing different types of invaders with ever more complicated attack formations.

Stats
See *Statistics*

Statistics (RPGs)
These specify the abilities of a particular character in the game. See *Attributes*.

STOS Basic
The original Game Creator on the ST. A direct ancestor of AMOS Basic.

Subroutine
Another name for a Gosub or Procedure, this is a small section of a Basic program designed to perform a specific task.

Synonyms (Adventures)
Words with an identical or similar meaning such as GET and TAKE. Lists of common synonyms are held for each of the possible commands in an adventure to reduce the risk of ambiguity.

Syntax errors
Errors generated by typing mistakes made when we are entering our program into the Amiga. E.g. `prunt "This is a syntax error".`

Systems analysis
The act of analysing a problem and generating a solution which often takes the form of a computer program.

Text plus adventures
A standard text adventure pepped up with a few pictures and a little sampled sound.

Text coordinates
These a special coordinates measured in units of a single character. They're used with the LOCATE command to position some text on the screen.

Thought experiments
Describes the process of thinking through the various implications of an idea before committing yourself to actually using it in one of our games. This can save us hours of frustration trying to write program which will never really work out in practice.

Tile
The small map elements used to generate the massive game maps required by an RPG or a scrolling arcade game. Each tile is assigned a number from 1 to 255. They are usually stored in the form of icons in Bank number 2, although we can also use Bobs, or screen blocks for this purpose.

Tile coordinates
A system of coordinates used for MAPS which works in units of a single tile. All coordinates are measure from the top left corner of the current MAP.

TOME
TOtal Map Editor. This is a separate utility from Shadow Software which allows us to generate massive playing areas in our games.

Two-phase scrolling
This is a system which scrolls a map display in two parts. The first Fine Scroller scrolls the display across an invisible map window, and the Course Scroller moves the map window through the map, in units of a single tile.

Vertical Scroller (Arcade games)
This is an arcade game set in a large vertical playfield which scrolls smoothly down the screen. The playfield can be generated using either the AMOS MAP definer or AMOS TOME.

X coordinate
The horizontal distance from the top of the screen, or map, to the current position. This distance can be measure in pixels (screen and hardware coordinates), characters (text coordinates), or tiles (map coordinates).

Y coordinate
The vertical distance from the top of the screen (or map) to the current position.

Zone
Areas of the screen defined using the AMOS Basic SET ZONE command. They can be used to create fancy control panels in our games.

INDEX

Amiga Game Maker's Manual

If you're interested in producing professional-looking games for the Amiga, all you need is AMOS Basic and this superb book. The creator of AMOS, François Lionette, designed it to be *the* most powerful games programming system for the Amiga. With AMOS, games equivalent to such best sellers as Xenon 2 and Blood Money can be produced – and in a fraction of the time you'd need if you were sweating it out with Assembler.

As you learn about AMOS, you'll be swept along by Stephen Hill's enthusiasm and depth of knowledge. All the major aspects of game design and programming are covered: Game planning – graphic design, mock-ups and sprites; Shoot-'em-ups – high-speed AMOS sprites, fire control, collision detection, background animation; Simulations – from simple 3D movement to flight simulators; Role playing – characters, scenarios and magic; Adventure games – plans, rooms and commands; Animation, scrolling, sound, and *much* more.

Numerous tested, working programs illustrate the book, and there is a disk available with a collection of AMOS programs – look inside the book for details.

Stephen Hill is the author of Sigma's best-selling *Game Maker's Manual for the Atari ST*. Stephen is one of the UK's acknowledged experts in this field – the author of numerous manuals and designer of dozens of Amiga games – so nobody is better placed to write this new book.

About Sigma Press:

We publish a wide range of books on all aspects of computing. Write or phone for a complete catalogue:

Sigma Press,
1 South Oak Lane, Wilmslow, Cheshire SK9 6AR

Phone: 0625 - 531035 Fax: 0625 - 536800

We welcome new authors.

SIGMA