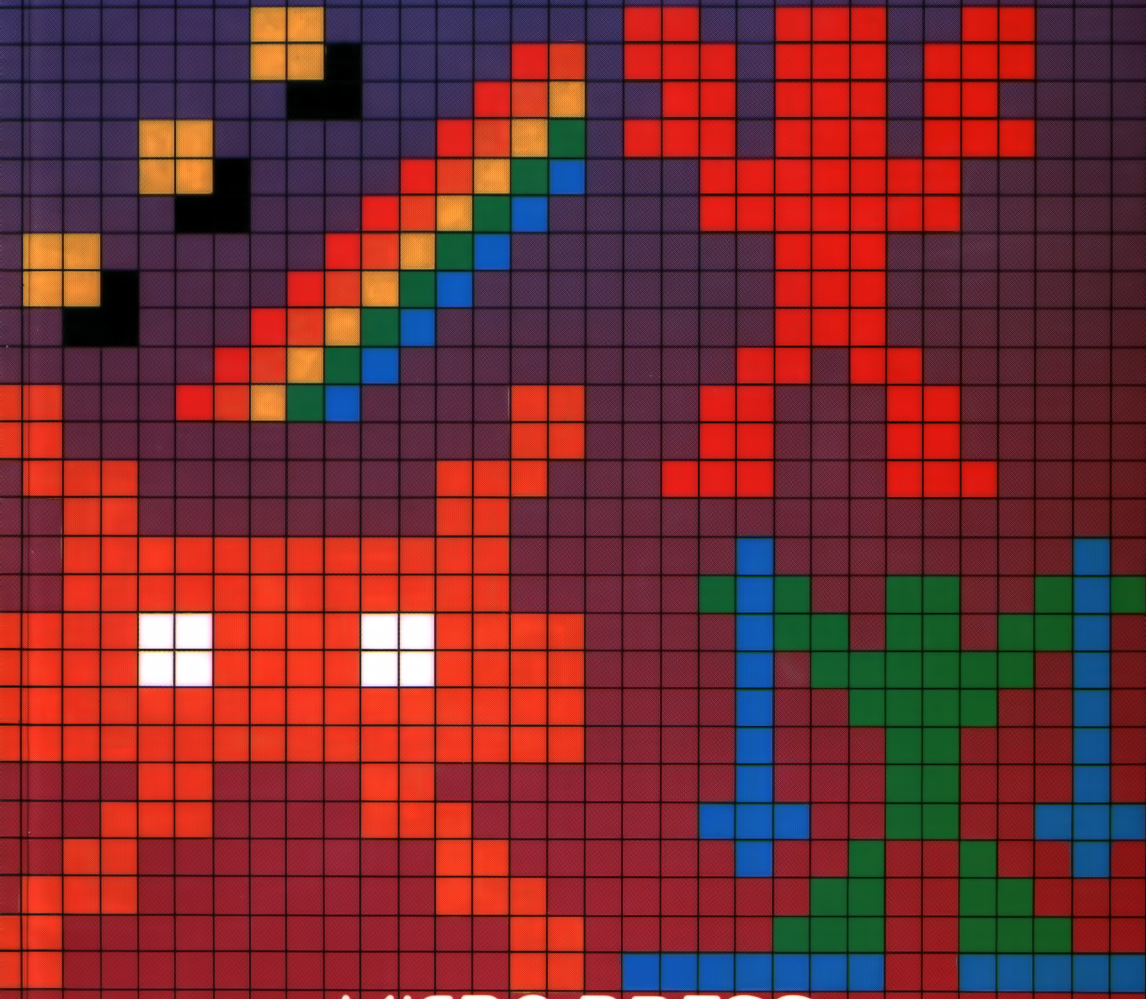
GRAPHICS PROGRAMMING TECHNIQUES ON THE AMSTRAD CPC 464

Wynford James



MICRO PRESS

Graphics Programming Techniques on the Amstrad CPC 464

Graphics Programming Techniques on the Amstrad CPC 464

Wynford James

MICRO PRESS

First published in 1985 in the United Kingdom by
Micro Press
Castle House, 27 London Road
Tunbridge Wells, Kent

British Library Cataloguing in Publication Data

James, Wynford
Graphics programming techniques on the
Amstrad CPC 464.
1. Amstrad CPC464 (Computer) —— Programming
2. Computer graphics
I. Title
001.64'43 QA76.8.A4

ISBN 0-7447-0027-2

Typeset by MC Typeset, Chatham, Kent
Printed and bound by Mackays of Chatham Ltd.

Contents

Introduction

This book will introduce you to some of the graphics programming techniques that you can use on the Amstrad CPC464. There are chapters on animation, both of characters and line-drawings, the production of graphs and bar charts, and pattern-drawing, to name a few.

Each chapter contains sample programs, many of them useful in their own right. For example, Chapter 2 contains a program which allows you to design your own characters on-screen and save them to a file; Chapter 3 includes routines to enable the simple construction and shading of pie-charts; Chapter 4 contains a drawing program that enables you to sketch on the screen, 'blow up' any part of the drawing and add detail, and save the resulting picture to a file.

Throughout the book I have assumed some knowledge of Basic and familiarity with loops, decisions and subroutines. Although beginners will enjoy using many of the programs as they stand, they will probably learn more from this book if they first read my previous publication, *Basic Programming on the Amstrad* (also available from Micro Press).

For the benefit of those who already have some experience of Basic but have not read my earlier book, Chapter 1 contains some material from that volume which serves to introduce the fundamental graphics commands available on the Amstrad.

Chapter One

Basic graphics

The screen display

In the real world there are many varieties of paper for different uses. An architect does not design houses on a notepad, and a novelist does not use foolscap paper to write stories. In computing the screen display is the equivalent to a sheet of paper, and it is useful to be able to change the display to suit the purpose.

The command `mode` followed by the number `0`, `1` or `2` selects one of the three screen displays allowed on the Amstrad. Each mode allows a different number of characters per line to be displayed on-screen, a different number of colours to be displayed simultaneously, and a different degree of graphics resolution (the 'fineness' with which lines can be drawn).

Mode	Number of lines	Characters per line
0	25	20
1	25	40
2	25	80

Figure 1.1 The three screen modes available on the Amstrad CPC 464.

Someone typing in a lot of text at the keyboard finds it useful to be able to see as much of it as possible on-screen. Mode 2 is best for this purpose — the Amstrad can print 25 lines with 80 characters in each line in mode 2.

The Amstrad automatically reverts to mode 1 when reset or switched on. Mode 1 has 25 lines with 40 characters per line. Mode 1 gives the most easily readable characters, and you can consider it as the 'working' mode when you are giving commands to the Amstrad. It is much easier to read text in mode 1!

Mode 0 gives 25 lines with just 20 characters per line. This mode is the best one to use if you want to produce colourful graphics, as it allows 16 different colours to be used on-screen simultaneously.

In any of the modes, text can be positioned anywhere on the screen by using text coordinates:

```
10 MODE 1
15 PRINT"Let's print beginning at (10,12
).",
20 LOCATE 10,12
30 PRINT"Here it is!"
```

The position (10,12) is referred to as a character position, and XX and YY are the TEXT COORDINATES for that position. The first number is called the X COORDINATE of the character (how far along it is) and the second number is the Y COORDINATE (how many lines down the screen it is). In this case, **LOCATE 10,12** causes the Amstrad to start printing at the 10th column along, and the 12th line down. You can use the command **CLS** to clear the screen after running the program.

Line 20 of the program tells the Amstrad to begin printing what follows at the specified text coordinates. The coordinates

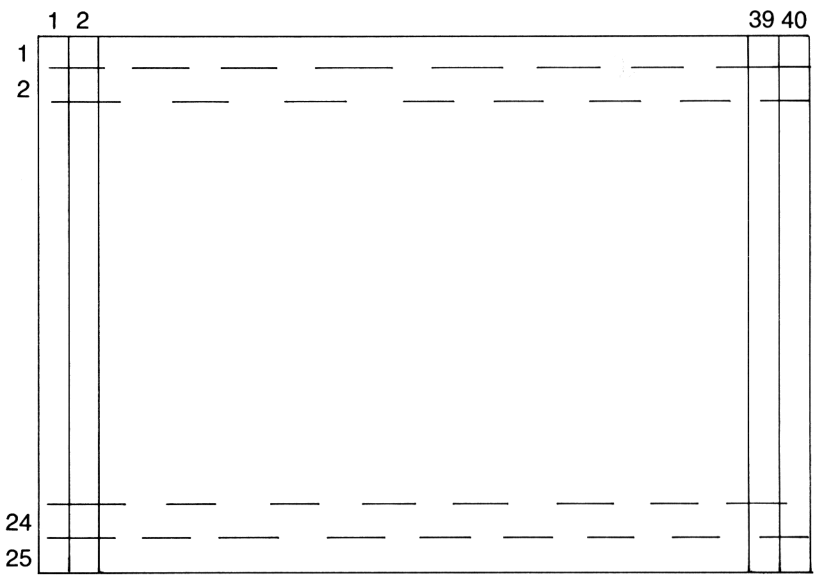


Figure 1.2 The screen display in mode 1.

vary according to the mode, as each mode can print a different number of characters on one 'line' on-screen.

Clearly, if we could only print at character positions the prospects for reasonable graphics would be pretty bleak. Mode 2 has 25 lines, each of 80 characters, but trying to construct a reasonable picture by printing characters to the appropriate position gives rather poor results. Fortunately, each character position can be subdivided still further, into smaller elements called PIXELS. Much finer lines can be drawn using individual pixels rather than character positions. Just as the number of characters per line varies from mode to mode, the size of a pixel varies from mode to mode.

However, the text coordinate system is inadequate to describe the location of pixels, because each character position is itself composed of several pixels. The Amstrad therefore uses a different coordinate system to describe pixel positions, and locates them by using GRAPHICS COORDINATES.

The graphics screen

The graphics coordinate system is coincident with the text coordinate system, but it does not operate in quite the same

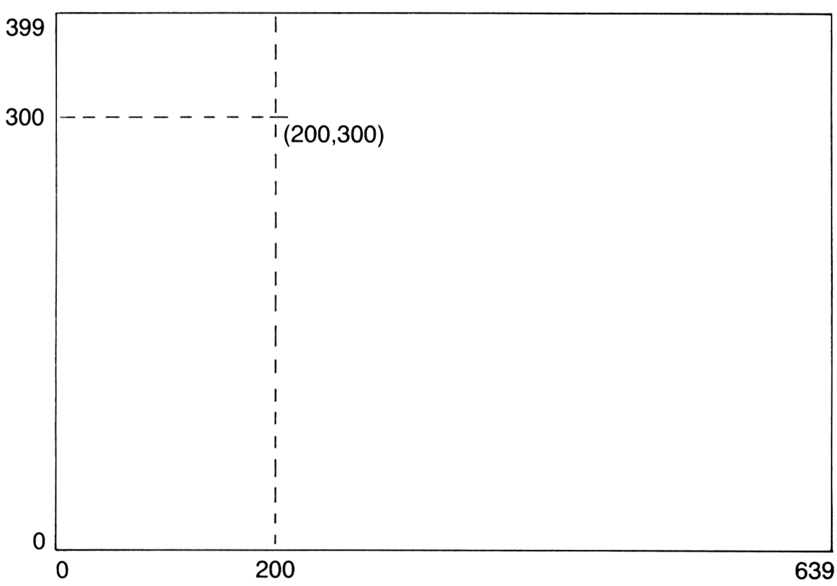


Figure 1.3 The graphics screen, showing the point (200,300).

way. The graphics screen is divided up into 640 points horizontally and 400 points vertically. We can identify the position of any point on the screen by describing how far along and how far up the screen the point is.

The position of the point in Figure 1.3 is (200,300). Notice that these GRAPHICS COORDINATES are measured from the bottom of the screen, and that the BOTTOM left hand point on the screen has the coordinates (0,0). This can be a bit confusing, because text coordinates work in a completely different way, with the TOP left hand character position having the coordinates (1,1)! Notice also that because the numbering of points begins with 0, the top right hand point has the coordinates (639,399) and NOT (640,400) as you might imagine.

The following program demonstrates two of the basic graphics commands available on the Amstrad:

```
10 MODE 1
20 MOVE 124,156
30 DRAW 300,300
40 DRAW 200,400
50 DRAW 124,156
```

Here we are using the GRAPHICS CURSOR to draw lines on the screen. Normally the graphics and text cursor remain together, but as soon as we use a graphics command the invisible graphics cursor is used.

The MOVE command in line 20 causes the graphics cursor to move invisibly to the point (124,156). The DRAW command makes the cursor move from its position at (124,156) to the new coordinates (300,300) drawing a line between the two points. The remaining DRAW commands in lines 40 and 50 draw the two other sides of a triangle.

In general terms, we can say that MOVE x,y causes the graphics cursor to move to the point x,y without drawing a line. DRAW x,y causes a line to be drawn from the last point visited by a MOVE or a DRAW to the point x,y. It is easy to draw quite complex pictures by storing the x and y coordinates for the points in DATA statements and then reading them:

```
10 MODE 0
20 x=1
```

```
29 REM draw picture with series of MOVEs
 and DRAWs
30 WHILE x>0
40 READ x,y
50 READ x1,y1
60 MOVE x,y
70 DRAW x1,y1
80 LOCATE 1,24:PRINT x;" ";y;:r$="":WHIL
E r$="":r$=INKEY$:WEND
90 WEND
100 END
110 DATA 308,162,344,174,344,174,360,216
,360,216,364,260,364,260,360,310
120 DATA 360,310,336,344,336,344,292,356
,292,356,248,352,248,352,212,342
130 DATA 212,342,200,316,200,316,196,246
,196,246,200,214,200,214,216,174
140 DATA 216,174,252,166,316,198,300,190
,300,190,272,190
150 DATA 272,190,256,198,244,200,244,210
,244,204,324,204,324,208,324,198
160 DATA 256,238,268,232,268,232,288,232
,288,232,296,242,276,242,276,284
170 DATA 288,284,300,296,300,296,324,286
,324,286,288,282,264,282,252,294
180 DATA 252,294,232,280,232,280,264,280
,196,280,180,296,180,296,172,266
190 DATA 172,266,192,236,364,236,384,266
,384,266,380,294,380,294,364,282
200 DATA 320,348,332,324,312,324,312,352
,288,354,292,326,272,354,280,326,264,348
210 DATA 224,344,216,334,264,344,268,228
,268,219,272,216,272,230,276,228,276,214
,280,214,280,230,280,230,284,212
220 DATA 288,212,284,230,288,228,288,220
230 DATA -1,-1
```

Many impressive effects can be produced simply by using the two statements **MOVE** and **DRAW**. Curves can be built up from straight lines by moving the ends of the lines by a fixed amount each time:

```
1 MODE 1
10 x=100:y=100
20 maximum=300
```

```
30 stepsize=10
40 FOR number=0 TO maximum STEP stepsize
50 MOVE x+number,y
60 DRAW x+maximum,y+number
70 MOVE x,y+number
80 DRAW x+number,y+maximum
90 NEXT
```

The resolution of the different modes

Although the graphics screen is divided into 640 horiontal and 400 vertical points, the Amstrad cannot really tell all these points apart. The graphics screen is the same for all the modes, but in some of the screen modes the Amstrad is better able to tell points apart than others. Run the above program again after editing line **10** to be:

```
1 MODE 0
```

The drawing remains the same, but the lines are much thicker and the picture looks more 'chunky'. Now try:

```
10 MODE 2
20 MOVE 200,300
30 DRAW 200,399
40 MOVE 201,200
50 DRAW 201,399
60 MOVE 202,100
70 DRAW 202,399
80 MOVE 203,0
90 DRAW 203,399
```

This time the lines are very fine. Mode 2 is called the HIGH RESOLUTION MODE, because when using mode 2 the Amstrad can distinguish between 640 points horizontally and 200 points vertically, which results in very fine lines when **DRAW** is used.

In mode 2 the Amstrad cannot tell the difference between points that are vertically too close. It would treat the points **(10,10)**and **(10,11)** as being exactly the same. In fact both mode 1 and mode 0 have the same vertical resolution of 200 points as mode 2, but their horizontal resolutions are much worse. Type:

```
10 MODE 1
20 MOVE 200,300
30 DRAW 200,399
40 MOVE 201,200
50 DRAW 201,399
60 MOVE 202,100
70 DRAW 202,399
80 MOVE 203,0
90 DRAW 203,399
```

and run the program again. Mode 1 is the MEDIUM RESOLUTION MODE, and in mode 1 the Amstrad can only show 320 separate horizontal points. This means that, for example, **(200,300)** and **(201,300)** are both treated as the same point. Now type:

```
10 MODE 0
```

and run the program for the third time. Mode 0 is the LOW RESOLUTION MODE, and can only identify 160 different horizontal points.

You may wonder why on earth anyone would choose to use a screen mode that produces 'chunky' drawings when the high resolution mode 2 is available. The main reason is that although mode 0 is low resolution, it can display drawings in up to 16 different colours on the screen at the same time. Modes 1 and 2 are much worse, as Figure 1.4 demonstrates.

Mode	Graphics resolution	Number of colours on-screen simultaneously
0	160×200	2
1	320×200	4
2	640×200	16

Figure 1.4 The different graphics resolutions and numbers of colours available in the different modes.

The Amstrad has a limited amount of memory. It can only record a certain amount of information about the screen in the RAM. As with many things in computing, there is a trade-off here. The RAM can be used to record details of many points of two possible colours, fewer points of four possible colours, or very few points with 16 possible colours. The Amstrad gives you the choice and you must select the mode which seems to suit your purposes best.

The PLOT statement

Each of the lines drawn in the previous programs was actually made up from a number of pixels. The Amstrad can display individual pixels on the screen, although a single pixel is rather difficult to see in mode 2! In fact each pixel is really made up of a number of points, but none of the modes is accurate enough to identify every point on the screen. A pixel in each of the modes is the smallest 'block' of points on screen that can be located in the different modes.

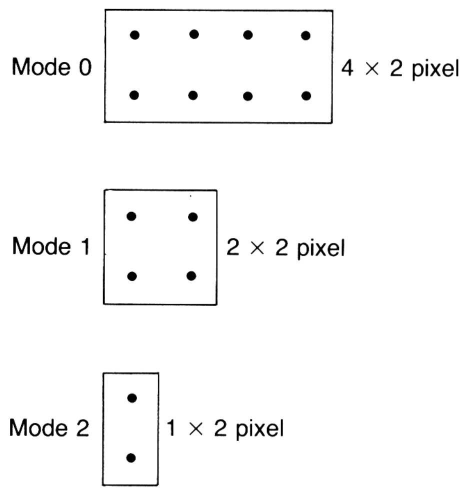


Figure 1.5 The size of a graphics pixel in each of the modes.

It may seem strange to have more points identified on the screen than can be displayed in any of the modes. The main reason for doing this is that it leaves room for future improvements in the graphics resolution without having to change the coordinate system completely.

PLOT works in the same way as **MOVE** or **DRAW** — the **PLOT** command must be followed by the x and y coordinates of the pixel to be plotted. This program plots six separate pixels on the screen:

```
10 MODE 0
20 PLOT 160,200
30 PLOT 320,200
40 PLOT 324,200
50 PLOT 328,200
```

```
60 PLOT 332,200
70 PLOT 480,200
```

In mode 0, the resolution is so low that the four pixels plotted in lines **30** to **60** merge to form a line, in mode 1 all the pixels can be seen, and in mode 2 the pixels are so fine that you may not be able to see them at all.

Adding colour

When the Amstrad is switched on, the micro is set to print

INK number	Colour
0	Black
1	Blue
2	Bright blue
3	Red
4	Magenta
5	Mauve
6	Bright red
7	Purple
8	Bright magenta
9	Green
10	Cyan
11	Sky blue
12	Yellow
13	White
14	Pastel blue
15	Orange
16	Pink
17	Pastel magenta
18	Bright green
19	Sea green
20	Bright cyan
21	Lime green
22	Pastel green
23	Pastel cyan
24	Bright yellow
25	Pastel yellow
26	Bright white

Figure 1.6 The 27 INK colours that can be used on the Amstrad CPC 464.

yellow text and graphics on a blue background in all the modes. In fact there are 27 different colours which can be displayed on the screen, although some of them are a bit difficult to tell apart. Each colour has a number, called the **INK** number, and whenever we refer to a colour we use this number rather than the name of the colour itself.

At this stage it is important to realise that the computer does not actually use the whole of the screen while it is printing or doing graphics. The Amstrad actually works within a large rectangle around which there is a border of unused screen. Although the Amstrad does not use this border, it is kept the same colour as the rest of the screen. The border is not really part of the computer memory because it is never used by the Amstrad for printing or drawing graphics.

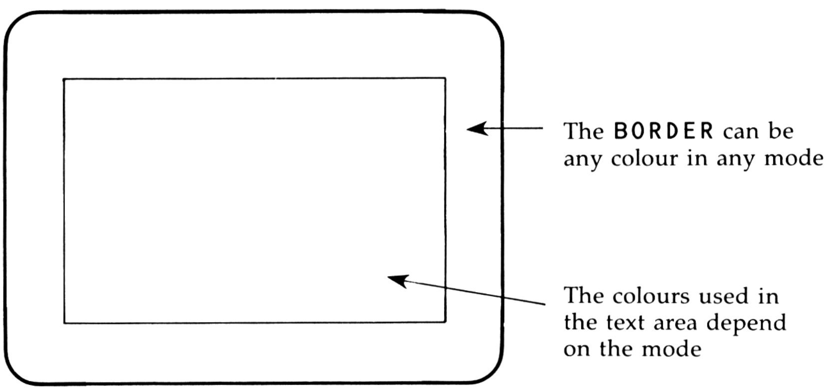


Figure 1.7 The **BORDER** area on your monitor or TV

The border can be set to be ANY colour in ANY mode. There are never any restrictions on the colour of the border. Mode 2 can only display two colours at once WITHIN the main screen rectangle, but its border can be ANY colour. Type:

```
10 MODE 2
20 BORDER 0
```

Refer back to Figure 1.6 and you will see that **0** is the **INK** number for the colour black. By typing **BORDER 0** you are telling the Amstrad to set a black border. Set the border to a few other colours — any number from 0 to 26 can be used, so there are 27 possible border colours altogether. **BORDER 26** gives a white border, for example.

The border can be set in modes 0 and 1 in exactly the same way. You will find that if you set a border and then change mode, the border remains set to its new colour. When the Amstrad is switched on or reset the border becomes blue, **BORDER 1**.

PEN and PAPER colours

The colours used within the main screen rectangle can also be changed. Here the question of RAM becomes important, and there are restrictions on the number of colours that can be displayed simultaneously on the screen at any one time.

We can change the colour the Amstrad 'writes' with by using the **PEN** command. Type:

```
MODE 0
PEN 4
```

From Figure 1.6 it might appear that this will give magenta characters, but the colours in the main screen work rather differently to those for the border! Choosing **PEN 4** actually causes the Amstrad to print in white. Think of **PEN 4** as being filled with white ink. Typing:

```
PEN 5
```

chooses a pen full of black ink. You can even have:

```
PEN 14
```

which gives you flashing blue/yellow ink!

There are 16 pens available for use in any mode and Figure 1.8 shows the colour number for the **INK** that the pens use. Note that the SAME pen can write with a DIFFERENT ink in another mode. This means that a program that works perfectly well in mode 0 may well give a blank screen in mode 2! The pen you have chosen may have the same colour as the background in mode 2. As you can see, the 16 pens aren't much use in mode 2, because 8 of them write in yellow and the other 8 in blue. We will see later how to change the inks that can be used in each mode.

The background colour can be changed as well by using the **PAPER** command. Reset the micro by holding down **[CTRL]** and **[SHIFT]** and pressing **[ESC]** then switch to mode 0. Type:

```
PAPER 3
```

and the next characters printed will be printed on a red background. The whole of the inner screen area can be changed to this new colour by using the **CLS** command. The Amstrad clears all of the main screen to the new paper colour.

PEN or PAPER number	Mode 0	Mode 1	Mode 2
0	1	1	1
1	24	24	24
2	20	20	1
3	6	6	24
4	26	1	1
5	0	24	24
6	2	20	1
7	8	6	24
8	10	1	1
9	12	24	24
10	14	20	1
11	16	6	24
12	18	1	1
13	22	24	24
14	1/24	20	1
15	16/11	6	24

Figure 1.8 The **PEN** and **PAPER** colours for the different modes. In mode 0, choosing **PEN** or **PAPER** 14 or 15 gives a flashing colour alternating between the two colours shown.

PAPER in mode 0 comes in the same 16 colours as the pens. **PEN 14** gave flashing blue/yellow ink, and **PAPER 14** gives a flashing blue/yellow background. Figure 1.6 can be used to help you select both the pen and paper colours. For example, to get red characters on a white background in mode 0, type:

```
INK 3
PAPER 4
CLS
```

PEN and **PAPER** commands can, of course, also be used in programs:

```
10 MODE 0
20 LOCATE 4,7
30 PEN 3
40 PAPER 5
```

```
50 PRINT "Red on black"
60 LOCATE 4,13
70 PEN 6
80 PAPER 3
90 PRINT "Blue on red"
100 LOCATE 4,19
110 PEN 5
120 PAPER 6
130 PRINT "Black on blue"
140 REM pen and paper back to normal
150 PEN 1
160 PAPER 0
```

Change line **10** and try running this program in the other two modes. You'll find you get some funny results, because the **PENs** contain different **INKs** in the other modes.

Here is another example:

```
10 MODE 0
20 redpeninmode0=3
30 blackpaperinmode0=5
40 PEN redpeninmode0
50 PAPER blackpaperinmode0
60 CLS
70 LOCATE 8,12
80 PRINT "Done!"
90 REM pen and paper back to normal
100 PEN 1
110 PAPER 0
```

The last two lines restore normal **PEN** and **PAPER** colours so you are not left with some unreadable mixture like yellow on white.

One frequent problem when playing around with the colours is that you can end up being unable to read anything on the screen, because the pen being used has the same colour ink as the background. As we have just seen, in a program this difficulty can be avoided by setting the pen colour back to normal before the program ends. Alternatively, set up one of the function keys on the Amstrad so that it restores normal **PEN** and **PAPER** colours when it is pressed:

```
KEY 128,CHR$(13)+"INK 0,1:INK 1,24"+CHR$(13)
```

Exercises

1) Write a program that selects a random point on the screen and draws a line to another random point in a random colour. The process is repeated from the new point and continues until 100 lines have been drawn.
2) Using **MOVE** and **DRAW** statements, draw a picture of a rocket. Give the rocket a name of your own choice which is printed along its length.
3) Print "**Different hues**" in the middle of the mode 0 screen, with every letter being in a different colour.

Graphics and colour

Drawing pictures with coloured lines is easy on the Amstrad. The commands **MOVE** and **DRAW**, used on their own, always result in lines drawn using **PEN 1** for whatever mode you are in. All the graphics programs so far have produced lines drawn with **PEN 1**. **PEN 1** contains **INK** number 24 in all modes, so the lines have all been bright yellow.

To get a different colour line, we must use an extension of the **DRAW** command. Reset the Amstrad, and type:

```
MOVE 100,100
DRAW 300,300,2
```

The Amstrad draws a line from **(100,100)** to **(300,300)** using **PEN 2**, which contains **INK** number 20, bright cyan, in mode 1. Type:

```
MOVE 300,300
DRAW 400,0,3
```

and a red line is drawn with **PEN 3** from (**300,300**) to (**400,0**). **PEN 3** uses **INK** number 6, red, in mode 1.

The commands are just as easy to use in a program. Again, remember that a program that works in one mode may not work in another because of the different **INK**s the **PEN**s have in different modes. This program draws a rectangle in mode 1, with one side in yellow, one in cyan, and the other two in red:

```
10 MODE 1
20 MOVE 100,100
30 DRAW 400,100
40 DRAW 400,300,3
```

```
50 DRAW 100,300,2
60 DRAW 100,100,3
```

Notice that at line 30 no PEN is specified, so the Amstrad automatically uses PEN 1, which draws a yellow line. After running the program once, run it again. You may be surprised to find that there is no longer a yellow line!

Whenever the Amstrad encounters a graphics command like DRAW, with no PEN specified, it will use the current PEN to obey the command. The first time the program is run, the Amstrad uses PEN 1 at line 30. After the program has been run, the last PEN used is PEN 3. This is now the current PEN colour, so when the program is run the second time, PEN 3 is used at line 30 where no PEN is specified. The advantage of this is that once PEN has been set in a draw command, all lines drawn after that are automatically drawn in that same colour unless a new PEN number is introduced:

```
10 MODE 1
20 MOVE 200,100
30 DRAW 400,200,2
40 DRAW 100,350
50 DRAW 200,100
```

Try running this program in mode 2, where PEN 2 holds a different colour INK.

Mode 0 is by far the best mode to use to produce a colourful graphics display if you are not too concerned about the resolution:

```
10 MODE 0
20 x=0:y=0
30 colour=1
40 FOR count=0 TO 350 STEP 4
50 MOVE x+count,y
60 DRAW x+350,y+count,colour
70 MOVE x,y+count
80 DRAW x+count,y+350
90 colour=(colour+1) MOD 16
100 NEXT
```

Changing the INK

So far we have only been able to see some of the colours that the Amstrad can produce. There are only 16 pens available, and yet Figure 1.6 shows that there are 27 INKs we can use. The Amstrad allows us to change the INK in each PEN so that we can choose any combination of colours for a particular mode.

The number of colours that can be used on-screen at the same time in any mode does NOT change, however. Although we can have bright red text on a white background in mode 2, these are the ONLY colours we could have on-screen at that time. We are ALWAYS limited to two colours in mode 2, four in mode 1, and 16 in mode 0.

When the Amstrad is switched on or reset, it reverts to mode 1 and uses PAPER Ø, which is blue (INK number 1) in all the modes, and PEN 1, which is yellow (INK number 24) in all the modes. Reset the computer now, and type:

```
INK 1,6
```

All the text on-screen changes colour from yellow to bright red instantly. The INK command needs two numbers. The first number is the number of the PEN or PAPER whose ink is to be changed. The second number gives the colour INK which is to be used instead.

The command INK 1,6 told the Amstrad to change the INK in PEN 1 to INK number 6, bright red. ANYTHING previously printed or drawn using PEN 1 has its colour changed from the old to the new INK. So to turn all the text blue type:

```
INK 1,2
```

and what was bright red now becomes bright blue. How would we return the text to normal? Perhaps you can work it out for yourself. Type:

```
INK 1,24
```

Normally PEN 1 uses INK 24 in all the modes, as you can see if you look back at Figure 1.8.

It is equally easy to change the PAPER colour. At the moment the Amstrad is using PAPER Ø, which is blue. Let's change this to white:

```
INK 0,26
```

Perhaps the text's a bit difficult to read. Try:

```
INK 0,6
```

or perhaps:

```
INK 0,0
```

The **PAPER** in all modes is usually blue, **INK** number **1**. Now try to turn everything back to normal yourself.

We don't need to have already used a **PEN** or **PAPER** colour to change it. Reset the Amstrad and type:

```
INK 3,0
```

Nothing SEEMS to happen. If we now go on to choose **PEN 3** in mode 1, Figure 1.8 suggests that text will be printed using **INK 6**, bright red. But we have just used the **INK** command to change the **INK** used by **PEN 3** to **INK 0**, black. Type:

```
PEN 3
```

and all the text is printed in black. Type:

```
INK 3,6
```

and now **PEN 3** and all the text it printed is set to **INK** number **6**, bright red. This colour change remains even if you change mode — try it.

It is even possible to set a colour so that it flashes between two different colours! Try:

```
INK 1,3,26
```

to see the text printed using **PEN 1** changing from **INK 3**, red, to **INK 26**, white, and back again.

A suitable selection of flashing **INKs** in a program can be used to give the illusion of movement. For example, we can set **PEN 1** to produce flashing yellow/red **INK**, and **PEN 2** to give flashing red/yellow **INK**. By printing alternate characters with alternate **PENs** we can give a 'rippling' effect which suggests the colour is moving along the line:

```
10 MODE 1
20 INK 1,3,12
30 INK 2,12,3
40 pencolour=1
50 FOR x=1 TO 40
```

```
60 IF pencolour=1 THEN pencolour=2 ELSE
pencolour=1
70 PEN pencolour
80 LOCATE x,13
90 PRINT CHR$(143);
100 NEXT
```

One obvious advantage of the **INK** command is that it enables us to choose any colour combinations from the 27 **INKs**. Even in a two colour mode like mode 2 we can brighten things up by using red text on a white background, instead of being restricted to just the colours yellow and blue which are available at switch-on. This program lets you see the more than 700 combinations of colour you now have in mode 2:

```
10 MODE 2
20 FOR x=0 TO 27
30 CLS
40 INK 0,x
50 FOR y=0 TO 27
60 IF x<>y THEN INK 1,y:PRINT "Ink ";y
70 response$=""
80 WHILE response$=""
90 response$=INKEY$
100 WEND
110 NEXT
120 NEXT
```

Exercises

1) Display your name in flashing characters on the screen. Choose the right colour background so the letters seem to appear and disappear.
2) Draw a picture of a fire using appropriate colours for the lines. (You may find it effective to make use of flashing colours.)
3) Draw a red crab lying on a sandy yellow beach. Choose the **PENs** and **INKs** so that the colours are the same no matter what mode is chosen when the program is run.

Codes and characters

The Amstrad CPC464 character set

The Amstrad can display a wide range of characters on-screen. As well as the familiar alphabetic and numeric characters, it can also produce crotchets, quavers, and even stick men, as you can see if you run this brief program:

```
10 MODE 1
20 FOR code=32 TO 255
30 PRINT CHR$(code);
40 NEXT
```

The Amstrad associates every character with a code number, called the ASCII code. This code number can range from 0 to 255. Codes 0 to 31 have special meanings to the computer, such as 'Move the cursor back one space' or 'Change the INK colour'. Codes 32 to 255 are for the lower and upper case alphabet, numbers, punctuation, etc. Line 30 of the program tells the Amstrad 'Print the character that has the following ASCII code'. Figure 2.1 shows some of the more useful ASCII codes.

Characters	ASCII codes
Various special codes	0–31
A space	32
0–9	48–57
A-Z	65–90
a-z	97–122

Figure 2.1 Some of the more useful ASCII codes.

The characters with codes 0 to 31 can be displayed if they are preceded by the character with an ASCII code of 1:

```
10 MODE 1
20 FOR code=0 TO 31
30 PRINT CHR$(1)CHR$(code);
40 NEXT
```

This clearly gives us a reasonable amount of choice when deciding which characters to use within a program. However, there are many circumstances where the Amstrad character set may not contain the character we need, for example in foreign language or mathematical work, or in a games program. In these cases we can use the facility available on the Amstrad to create our own USER-DEFINED CHARACTERS. Before we find out how to do this, it will be useful to look at how the Amstrad stores characters and why this method of storage limits us to only 256 predefined characters with ASCII codes 0 to 255.

Bits, bytes and binary

How does the Amstrad store information? To look at it in a simplified way, we can view the computer as containing thousands of switches, each of which can be set to be 'on' or 'off'. In the Amstrad, these 'switches' are set together in blocks of eight. If we represent an 'off' switch by Ø and an 'on' switch by 1, we can show all the possible combinations of 'switches' as in Figure 2.2.

```
Ø Ø Ø Ø Ø Ø Ø Ø
Ø Ø Ø Ø Ø Ø Ø 1
Ø Ø Ø Ø Ø Ø 1 Ø
Ø Ø Ø Ø Ø Ø 1 1
        .
        .
        .
1 1 1 1 1 1 Ø 1
1 1 1 1 1 1 1 Ø
1 1 1 1 1 1 1 1
```

Figure 2.2 The 256 possible combinations of 8 'switches'.

You will perhaps not be surprised to see that there are altogether 256 ways in which the 'switches' can be set. Each of the numbers is a BINARY NUMBER, composed only of the digits 0 and 1. The digits are binary digits, or BITS for short. Any combination of 8 bits is referred to as a BYTE.

Binary is a way of counting in twos, just as we count in tens, and each of the binary numbers is equivalent to a number in our own system of counting. It is relatively easy to convert any binary number into the more familiar decimal numbers, as we shall see in a moment.

The byte is the fundamental unit of information storage on the Amstrad. Many of the limitations of the machine arise because an 8-bit byte can be 'set' in just 256 different ways. Only 256 predefined characters are provided, because each character can be given an ASCII reference code that can be stored in a single byte. A program line can be from 0 to 255 characters long, as its length is stored in one byte. If 257 predefined characters were provided, or a line could be longer than 255 characters, the information would have to be stored in two bytes, and this would use up a lot of extra memory on the computer.

Each of the predefined characters has an ASCII code associated with it, but this in itself is not enough to enable the Amstrad to produce the required character on the screen. Every character is built up on an 8 × 8 grid, as for example the upper case 'A'.

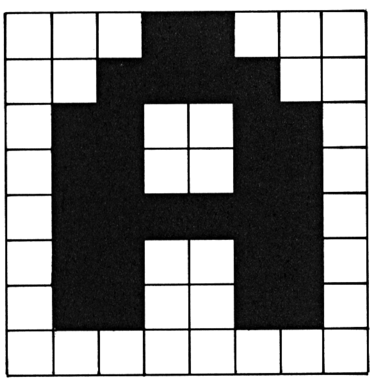


Figure 2.3 An upper case letter 'A'.

Each square on the grid will be either 'on' or 'off' (lit or unlit) when displayed on the screen . . . sounds like binary numbers, doesn't it? And with good reason, because the 8 × 8 grid is another consequence of the 8-bit byte. Each row of the grid can be stored as a byte, and the complete description of an entire character can thus be stored in 8 bytes.

```
0 0 0 1 1 0 0 0
0 0 1 1 1 1 0 0
0 1 1 0 0 1 1 0
0 1 1 0 0 1 1 0
0 1 1 1 1 1 1 0
0 1 1 0 0 1 1 0
0 1 1 0 0 1 1 0
0 0 0 0 0 0 0 0
```

Figure 2.4 The 8-byte binary character definition of the letter 'A'.

Binary numbers are cumbersome to work with, because they are lengthy and it is easy to insert or delete extra 0s or 1s by mistake. The binary values for the bytes can be converted to decimal simply by adding together the figures at the top of the column for any squares in a row that are shaded. The Amstrad

```
1
2 6 3 1
8 4 2 6 8 4 2 1

0 0 0 1 1 0 0 0                   16+8 =  24
0 0 1 1 1 1 0 0              32+16+8+4 =  60
0 1 1 0 0 1 1 0              64+32+4+2 = 102
0 1 1 0 0 1 1 0              64+32+4+2 = 102
0 1 1 1 1 1 1 0        64+32+16+8+4+2 = 126
0 1 1 0 0 1 1 0              64+32+4+2 = 102
0 1 1 0 0 1 1 0              64+32+4+2 = 102
0 0 0 0 0 0 0 0                        =   0
```

Figure 2.5 The character definition of the letter 'A' using decimal numbers.

can simplify this process considerably by doing the work for you! The command **PRINT STR$(number)** will convert a number into its decimal string equivalent. Binary numbers must begin with '**&X**', otherwise the Amstrad will take the number to be a very large decimal number that happens to be made up of 0s and 1s! Let's confirm that our calculations above were correct:

```
10 MODE 1
20 number=1
30 WHILE number>0
40 INPUT "Input the binary number, prece
ded by &X. ",number
50 PRINT "This is the decimal number "ST
R$(number)
60 WEND
```

You may need occasionally to convert decimal numbers to binary. The Amstrad will do this for you as well:

```
10 MODE 1
20 number=1
30 WHILE number>0
40 INPUT "Input the decimal number ",num
ber
50 PRINT "This is the binary number "BIN
$(number)
60 WEND
```

A point to note with both these conversions is that the end result is a STRING. You cannot carry out arithmetic on strings, and if you wish to do so you will first have to convert the string into a number:

```
10 MODE 1
20 number=1
30 WHILE number>0
40 INPUT "Input the decimal number ",num
ber
50 PRINT "This is the binary number "BIN
$(number)
54 REM VAL converts a string to a numeri
c value
55 numeric=VAL(BIN$(number))
56 PRINT "This is the number ",numeric
60 WEND
```

Defining your own characters

We now know the values for the eight bytes the Amstrad uses to describe the letter 'A'. Any of the 16 characters with ASCII codes 240 to 255 can automatically be redefined, so let's change character 240 to the letter 'A':

```
10 MODE 1
20 REM SYMBOL defines character
30 SYMBOL 240,24,60,102,102,126,102,102,
0
40 PRINT CHR$(240)
```

SYMBOL in line 30 tells the Amstrad we want to define a new character. The first number, 240, gives the ASCII code of the character, and the eight numbers following define each 'row' on the character grid.

If we need to redefine more than 16 ASCII codes, we must use the **SYMBOL AFTER** statement:

```
10 MODE 1
20 SYMBOL AFTER 65
30 SYMBOL 65,231,195,153,153,129,153,153
,255
40 PRINT CHR$(65)
```

Line 20 tells the Amstrad that we wish to be able to redefine any ASCII code of 65 or greater. Line **30** redefines ASCII code 65, for the uppercase **A**, so that points that were lit become unlit, and vice versa, as you can see if you try a capital A! The previous character definition has now been lost. It can be regained either by resetting the machine or, less drastically, by using another **SYMBOL AFTER** statement, at which point all characters are reset to their original definitions:

```
10 MODE 1
20 SYMBOL AFTER 65
30 SYMBOL 65,231,195,153,153,129,153,153
,255
40 PRINT CHR$(65)
50 SYMBOL AFTER 70
```

Hexadecimal

We have seen that the Amstrad can easily convert binary numbers to decimal to make life easier. Although we are all familiar with decimal numbers, in computing it has become traditional to use the HEXADECIMAL system, which involves counting in 16s.

Hex numbers are usually written preceded by '**&**' to avoid any confusion with decimal numbers. It is worth getting used to hex. The value of any byte can be shown using just two characters in the hexadecimal system. We could describe the letter A just as easily using hexadecimal numbers:

```
10 MODE 1
20 SYMBOL 240,&18,&3C,&66,&66,&7E,&66,&6
6,0
30 PRINT CHR$(240)
```

Decimal number	Hexadecimal equivalent
0	0
1	1
.	.
.	.
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11
.	.
30	1E
.	.
100	64
.	.
255	FF

Figure 2.6 Some decimal numbers and their hexadecimal equivalents.

Because the Amstrad works with bytes many apparently meaningless decimal numbers assume significance if they are shown in hex. For example, you will find that the Amstrad will reject any line number greater than 65535. This seems a quite arbitrary decimal number, but when converted into hex it becomes &FFFF and the reason for the restriction becomes clear: 65535 is the largest number that will fit into two bytes. The use of two bytes means that there are 256 × 256 = 65536 different line numbers available, from 0 to 65535. Allowing greater line numbers would require three bytes to store every line number, and storing a line number as a single byte would restrict us to line numbers from 0 to 255.

The Amstrad makes it easy to convert from decimal to hex:

```
10 MODE 1
20 number=1
30 WHILE number>0
40 INPUT "Input the decimal number ",num
ber
50 PRINT "In hexadecimal this is "HEX$(n
umber)
60 WEND
```

Conversion from hex to decimal involves the use of **PRINT STR$(number)** again, but this time to signify that the number concerned is a hex number it is preceded by '**&**':

```
10 MODE 1
20 number=1
30 WHILE number>0
40 INPUT "Input the hexadecimal number,
preceded  by & ",number
50 PRINT "This is the decimal number ",S
TR$(number)
60 WEND
```

Games

User-defined characters really come in useful in games programs. This program defines a 'dog' character which is then printed in a variety of random positions on the screen:

```
10 MODE 0
20 SYMBOL 240,0,4,7,132,124,130,130,0
30 FOR randomdogs=1 TO 30
40 randomx=INT(19*RND(1)+1)
50 randomy=INT(24*RND(1)+1)
60 pencolour=INT(15*RND(1)+1)
70 PEN pencolour
80 LOCATE randomx,randomy
90 PRINT CHR$(240)
100 NEXT
```

To move the dog around on-screen we must print a space at its present position to erase the old character, and then print to the new position:

```
10 MODE 0
20 SYMBOL 240,0,4,7,132,124,130,130,0
30 dog$=CHR$(240)
40 PEN 1
50 dogx=13:dogy=10
60 response$=""
70 WHILE response$<>"e"
80 newy=dogy:newx=dogx
90 response$=INKEY$
100 IF response$="a" AND dogy>1 THEN new
y=dogy-1
110 IF response$="z" AND dogy<25 THEN ne
wy=dogy+1
120 IF response$="," AND dogx>1 THEN new
x=dogx-1
```

```
130 IF response$="." AND dogx<20 THEN ne
wx=dogx+1
140 IF dogx<>newx OR dogy<>newy THEN LOC
ATE dogx,dogy:PRINT " ";:dogx=newx:dogy=
newy
150 LOCATE dogx,dogy
160 PRINT dog$
170 WEND
```

Animation can be made more interesting by defining a series of characters, each of which is only printed if movement takes place in a particular direction. We can modify the above program to illustrate this, although rather than define four new characters I have used the predefined characters with ASCII codes 240 to 243. These are all arrows, pointing in different directions:

```
10 MODE 0
30 arrow$=CHR$(240)
40 PEN 1
50 arrowx=13:arrowy=10
60 response$=""
70 WHILE response$<>"e"
80 newy=arrowy:newx=arrowx
90 response$=INKEY$
100 IF response$="a" AND arrowy>1 THEN n
ewy=arrowy-1:arrow$=CHR$(240)
110 IF response$="z" AND arrowy<25 THEN
newy=arrowy+1:arrow$=CHR$(241)
120 IF response$="," AND arrowx>1 THEN n
ewx=arrowx-1:arrow$=CHR$(242)
130 IF response$="." AND arrowx<20 THEN
newx=arrowx+1:arrow$=CHR$(243)
140 IF arrowx<>newx OR arrowy<>newy THEN
 LOCATE arrowx,arrowy:PRINT " ";:arrowx=
newx:arrowy=newy
150 LOCATE arrowx,arrowy
160 PRINT arrow$
170 WEND
```

The above skeleton program can serve as the basis for many games.

One of the problems with user-defined characters is that it is a tedious business designing characters by hand, and often the result when displayed on the screen is very different to the planned effect. The character-designing program on the Amstrad 'welcome' tape suffers from a major defect in that it fails to display the **SYMBOL** definition needed to recreate a character.

The following program enables you to create characters on-screen. It displays the **SYMBOL** definition needed to produce the character and gives you the option of saving that data to a file. This gives you the opportunity of setting up your own 'library' of user-defined characters which can be used in future programs:

```
10 MODE 1
20 DEFINT c,n,x,y
29 REM set up arrays to hold character c
ode
30 DIM code(8,8),symb(8)
40 INK 3,20,6
50 name$="??????"
60 number=240
69 REM display 'empty' character
70 GOSUB 1000
80 GOSUB 3000:GOSUB 4000
89 REM scan keyboard for response
90 response$=""
100 WHILE response$<>"e"
110 newx=x:newy=y
120 response$=LOWER$(INKEY$)
129 REM next four lines move cursor up/d
own/left/right
130 IF response$="a" AND y>starty THEN n
ewy=y-1
140 IF response$="z" AND y<starty+7 THEN
 newy=y+1
150 IF response$="," AND x>startx THEN n
ewx=x-1
160 IF response$="." AND x<startx+7 THEN
 newx=x+1
169 REM update position if necessary
170 IF newx<>x OR newy<>y THEN GOSUB 200
0
179 REM change colour of point on depres
sion of space bar
180 IF response$=" " THEN GOSUB 7000:GOS
```

```
UB 3000
189 REM output symbol definition to file
 if 'o' pressed
190 IF response$="o" THEN GOSUB 5000:GOS
UB 1000:GOSUB 3000:GOSUB 4000
199 REM input symbol definition from fil
e if 'i' pressed
200 IF response$="i" THEN GOSUB 6000:GOS
UB 3000:GOSUB 4000
210 PEN 14
220 LOCATE x,y
230 PRINT CHR$(203);
240 WEND
250 END
999 REM empty symbol definition
1000 CLS
1010 SYMBOL number,0,0,0,0,0,0,0,0
1020 PEN 1
1030 FOR count=1 TO 8
1040 FOR count1=1 TO 8
1050 code(count,count1)=1
1060 NEXT
1070 NEXT
1079 REM print 8x8 empty squares to repr
esent blank character
1080 startx=2:starty=2
1090 FOR x=startx TO startx+7
1100 FOR y=starty TO starty+7
1110 LOCATE x,y
1120 PRINT CHR$(233);
1130 NEXT
1140 NEXT
1150 x=startx:y=starty
1160 RETURN
1999 REM print correct colour character
at old position on cursor move
2000 LOCATE x,y
2010 codex=x-startx+1:codey=y-starty+1
2020 PEN code(codex,codey)
2030 PRINT CHR$(233);
2040 x=newx:y=newy
2050 RETURN
2999 REM convert code array to 8 decimal
 numbers for SYMBOL definition
3000 FOR count=1 TO 8
```

```
3010 symb$="&X"
3020 FOR count1=1 TO 8
3030 symb$=symb$+MID$(STR$(code(count1,c
ount)-1),2,1)
3040 NEXT
3050 symb(count)=VAL(symb$)
3060 NEXT
3070 SYMBOL number,symb(1),symb(2),symb(
3),symb(4),symb(5),symb(6),symb(7),symb(
8)
3080 PEN 1
3090 LOCATE 1,starty+10
3100 PRINT "Symbol is: ";CHR$(number);
3110 LOCATE 1,starty+12
3120 PRINT "SYMBOL ";number;symb(1);symb
(2);symb(3);symb(4);symb(5);symb(6);symb
(7);symb(8);
3130 RETURN
4000 PEN 1
4010 LOCATE 1,starty+15
4020 PRINT "Symbol name: ";name$
4030 LOCATE 1,starty+17
4040 PRINT "Symbol number: ";number
4050 RETURN
4999 REM save character definition to fi
le
5000 LOCATE 1,21
5010 PEN 1
5020 INPUT "Name of symbol";name$
5030 OPENOUT name$
5040 WRITE #9,name$,number,symb(1),symb(
2),symb(3),symb(4),symb(5),symb(6),symb(
7),symb(8)
5050 CLOSEOUT
5059 REM update SYMBOL name and number f
or next definition
5060 name$="??????"
5070 number=number+1
5080 RETURN
5999 REM input character definition from
 file
6000 LOCATE 1,21
6010 PEN 1
6020 INPUT "Name of symbol";name$
6030 OPENIN name$
```

```
6040 INPUT #9,name$,number,symb(1),symb(
2),symb(3),symb(4),symb(5),symb(6),symb(
7),symb(8)
6050 CLOSEIN
6060 CLS
6069 REM turn symb array into binary for
 conversion to code array
6070 FOR count=1 TO 8
6080 symb$=BIN$(symb(count))
6090 length=LEN(symb$)
6100 symb$=STRING$(8-length,"0")+symb$
6110 FOR count1=1 TO 8
6120 LOCATE startx+count1-1,starty+count
-1
6130 code=VAL(MID$(symb$,count1,1))+1
6140 PEN code
6150 PRINT CHR$(233);
6160 code(count1,count)=code
6170 NEXT
6180 NEXT
6190 RETURN
6999 REM toggle colour at cursor positio
n on depression of space bar
7000 codex=x-startx+1:codey=y-starty+1
7010 IF code(codex,codey)=1 THEN code(co
dex,codey)=2 ELSE code(codex,codey)=1
7020 RETURN
```

Exercises

1) Design a 'spider' character and write a program that redefines the full stop key to produce your arachnid friend.
2) Write a program to enable you to move your spider about on-screen. You might like to use some of the pattern-drawing routines from the last chapter to produce some suitable webbing.
3) Improve the previous program by designing up-, down-, left-, and right-facing spiders, and print the correct character when movement is made in a particular direction.

Multiple characters

In mode 1 a single character is not very large, and it may be preferable to create a larger figure built up from several shapes. The individual characters can be joined together to make up a single string if the figure is completely horizontal:

```
10 MODE 1
19 REM define 3 characters for lorry
20 SYMBOL 240,0,0,96,96,96,127,18,12
30 SYMBOL 241,0,0,0,0,0,255,0,0
40 SYMBOL 242,248,132,132,255,255,255,72
,48
50 lorry$=CHR$(240)+CHR$(241)+CHR$(242)
60 LOCATE 18,13
70 PRINT lorry$
```

The 'lorry' character can easily be placed under the control of the keyboard, although for the sake of realism let's just move it in one direction only — to the right:

```
10 MODE 1
20 REM define 3 characters for lorry
30 SYMBOL 240,0,0,96,96,96,127,18,12
40 SYMBOL 241,0,0,0,0,0,255,0,0
50 SYMBOL 242,248,132,132,255,255,255,72
,48
60 lorry$=CHR$(240)+CHR$(241)+CHR$(242)
70 x=1:y=13
80 LOCATE x,y
90 PRINT lorry$
100 response$=""
109 REM scan keyboard until 'e' pressed
110 WHILE response$<>"e"
120 newx=x
130 response$=LOWER$(INKEY$)
140 IF response$="." THEN newx=x+1
149 REM if lorry has moved print a space
 where the left end was
150 IF newx<>x THEN LOCATE x,y:PRINT " "
;lorry$
160 x=newx
170 WEND
```

Notice what happens when the figure gets too close to the right-hand edge: the whole figure is automatically printed at the start of the next line. The Amstrad will not print any character that moves the text cursor outside the text window. If

ASCII code	Action
8	Cursor moves back a character
9	Cursor moves forward one character
10	Cursor moves down a line
11	Cursor moves up a line

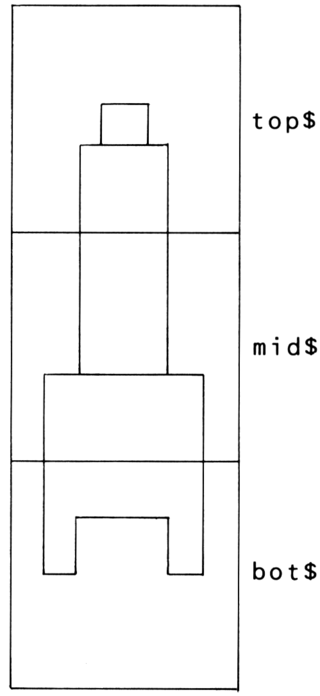
Figure 2.7 The four ASCII codes for cursor movement.

a character does fall into one of these positions, the computer moves the text cursor to an allowed position, using the following rules:

1) If the cursor moves beyond the right edge of the screen, it is moved to the first position on the next line.
2) If the cursor moves beyond the left edge of the screen, it is moved to the last position on the previous line.
3) If the cursor goes off the top of the screen, the screen scrolls down a line, and the cursor remains on the new top line.
4) If the cursor goes off the bottom of the screen, the screen scrolls up a line, and the cursor remains on the new bottom line.

Although only the front of the lorry falls into an illegal category, the computer is printing the lorry as a single string, and regards the whole string as being printed at an illegal position. Consequently as soon as the lorry reaches an x text coordinate of 39, which would make the front 'poke out' into an illegal cursor position, the Amstrad prints the ENTIRE string at the beginning of the next line. This is important to remember, as it means that the size of a multiple character affects the positions at which it can be safely printed on-screen. In this case, the maximum x text coordinate that can be used is 38.

Printing a vertical figure might seem more difficult, because surely each part of the figure will need a different **LOCATE** statement to print it? Here we can take advantage of the fact that four of the lower ASCII codes do nothing but move the cursor in particular directions. By including cursor move commands we can describe a vertical figure by a single string. These cursor move characters are not printed by the Amstrad



The 'rocket' could be printed as a single string composed of:

```
top$ + CHR$(8) + CHRS(8) + CHR$(10)
       \___________________________/
             moves cursor back and down
             ready to print next character
+ mid$ + CHR$(8) + CHR$(8) + CHR$(10)
         \___________________________/
             same cursor movement again

+ bot$
```

Figure 2.8 A vertical multiple character created using cursor moves.

but serve only as instructions to the text cursor where to move next.

The 'rocket' can be printed to the screen using a single **LOCATE** statement:

```
10 MODE 1
19 REM define 3 characters for rocket
20 SYMBOL 240,0,24,24,24,24,36,36,36
30 SYMBOL 241,36,36,36,36,36,36,36,36
40 SYMBOL 242,66,129,129,129,129,153,195
,129
```

```
50 rocket$=CHR$(240)+CHR$(8)+CHR$(10)+CH
R$(241)+CHR$(8)+CHR$(10)+CHR$(242)
60 x=20:y=20
70 LOCATE x,y
80 PRINT rocket$
90 response$=""
100 REM scan keyboard until 'e' pressed
110 WHILE response$<>"e" AND y>1
120 newy=y
130 response$=LOWER$(INKEY$)
140 IF response$="a" THEN newy=y-1
150 REM if rocket has moved print a spac
e where the bottom was
160 IF newy<>y THEN LOCATE x,y+2:PRINT "
 ":LOCATE x,newy:PRINT rocket$
170 y=newy
180 WEND
```

Unfortunately the Amstrad displays a peculiar habit when dealing with strings containing cursor moves. The above figure is clearly vertical, and no part of it lies in an illegal position. However, the computer considers it to be a string 7 characters long and hence will not allow the 'rocket' to be printed to any x text coordinate greater than 34. You can demonstrate by changing line **60** to:

```
10 MODE 1
19 REM define 3 characters for rocket
20 SYMBOL 240,0,24,24,24,24,36,36,36
30 SYMBOL 241,36,36,36,36,36,36,36,36
40 SYMBOL 242,66,129,129,129,129,153,195
,129
50 rocket$=CHR$(240)+CHR$(8)+CHR$(10)+CH
R$(241)+CHR$(8)+CHR$(10)+CHR$(242)
60 x=35:y=20
70 LOCATE x,y
80 PRINT rocket$
90 response$=""
100 REM scan keyboard until 'e' pressed
110 WHILE response$<>"e" AND y>1
120 newy=y
130 response$=LOWER$(INKEY$)
140 IF response$="a" THEN newy=y-1
150 REM if rocket has moved print a spac
e where the bottom was
```

```
160 IF newy<>y THEN LOCATE x,y+2:PRINT "
 ":LOCATE x,newy:PRINT rocket$
170 y=newy
180 WEND
```

We can use cursor moves to create more complex figures, but it is important to be aware of the restrictions this places on positioning. For all but the simplest of figures it is probably better to print by a series of **LOCATE** statements. This brief program demonstrates the two approaches:

```
10 MODE 1
19 REM define 3 characters for rocket
20 SYMBOL 240,0,24,24,24,24,36,36,36
30 SYMBOL 241,36,36,36,36,36,36,36,36
40 SYMBOL 242,66,129,129,129,129,153,195
,129
49 REM define using cursor moves
50 rocket$=CHR$(240)+CHR$(8)+CHR$(10)+CH
R$(241)+CHR$(8)+CHR$(10)+CHR$(242)
59 REM this way we can't use xcoordinate
s greater than 34
60 x=1:y=20
70 LOCATE 3,23
80 PRINT "Character defined using cursor
 moves"
90 FOR ycoord=y TO 1 STEP -1
100 LOCATE x,ycoord+3
110 PRINT " "
120 LOCATE x,ycoord
130 PRINT rocket$
140 NEXT
149 REM wait for key depression before c
ontinuing demo
150 response$=""
160 WHILE response$=""
170 response$=LOWER$(INKEY$)
180 WEND
190 CLS
199 REM this way we can print to ANY x c
oordinate
200 x=36:y=20
210 LOCATE 7,23
220 PRINT "Character defined using LOCAT
E for each part"
```

```
229 REM print each part of the rocket wi
th separate LOCATE statements
230 FOR ycoord=y TO 1 STEP -1
240 LOCATE x,ycoord
250 PRINT CHR$(240)
260 LOCATE x,ycoord+1
270 PRINT CHR$(241)
280 LOCATE x,ycoord+2
290 PRINT CHR$(242)
299 REM delete old rocket base
300 LOCATE x,ycoord+3
310 PRINT " "
320 NEXT
```

More interesting effects can be achieved if two slightly differing figures are defined, and then displayed alternately. We can use this idea to produce a 'snake' which wriggles its way across the screen:

```
10 MODE 1
19 REM define 2 'snake' characters
20 SYMBOL 240,0,0,32,80,81,74,130
30 SYMBOL 241,0,0,0,132,74,81,80,32
40 snake1$=CHR$(240)
50 snake2$=CHR$(241)
60 snake$=snake1$
70 y=13
80 FOR x=1 TO 39
89 REM toggle printing of character betw
een one snake and the other
90 IF snake$=snake1$ THEN snake$=snake2$
 ELSE snake$=snake1$
100 LOCATE x,y
109 REM print space to delete snake's ta
il
110 PRINT " ";snake$
119 REM delay - otherwise it's all over
too quickly!
120 oldtime=TIME
130 WHILE TIME<oldtime+10
140 WEND
150 NEXT
```

We could similarly define two 'lorry' images which vary slightly to give the impression that the vehicle is jolting its way along. Or we could make the earlier 'dog' wag its tail as it strolls about:

```
10 MODE 1
19 REM define 2 'dog' characters
20 SYMBOL 240,0,132,135,132,124,130,130,
0
30 SYMBOL 241,0,36,71,132,124,130,65,0
40 dog1$=CHR$(240)
50 dog2$=CHR$(241)
60 dog$=dog1$
70 y=13
80 FOR x=1 TO 39
89 REM toggle printing of character betw
een one dog and the other
90 IF dog$=dog1$ THEN dog$=dog2$ ELSE do
g$=dog1$
100 LOCATE x,y
109 REM print space to delete old dog
110 PRINT " ";dog$
119 REM delay - otherwise it's all over
too quickly!
120 oldtime=TIME
130 WHILE TIME<oldtime+30
140 WEND
150 NEXT
```

Exercises

1) Create your own multiple-character version of a bus, and drive it across the screen.
2) Design two circular characters with a cross-piece at differing angles, and print them alternately to the screen to give the impression of a rolling wheel.
3) Add the rolling wheels to your bus as it is moved across the screen.

Improving the resolution

Although a number of amusing games can be devised using only the text screen, there are clearly limitations imposed. The best text resolution is available in mode 2, and even this has just 25 lines of 80 characters. In contrast, the worst graphics resolution is 160 by 200 points. Fortunately, the Amstrad allows text characters to be printed to a graphics position, and this means we can produce smoother animation, and give the user finer control in games. More seriously, it means that graphs and charts can be accurately labelled at any point rather than at the closest text position.

The switch from text to graphics coordinates brings other changes as well. After a **TAG** (Text At Graphics) command has been given, characters can no longer be printed following a **LOCATE** statement. Instead the graphics command **MOVE** must be used to position the graphics cursor. The text character is tagged to the cursor by its TOP LEFT corner. This program moves a single character 'Space Invader' to demonstrate the principle:

```
10 MODE 1
19 REM define space invader
20 SYMBOL 240,24,60,126,219,255,255,165,
165
30 invader$=CHR$(240)
40 graphicsx=100:graphicsy=200
49 REM join text to graphics cursor
50 TAG
60 FOR x=graphicsx TO 600
70 MOVE x,graphicsy
78 REM print space to rub out old invade
r first
79 REM omit the semi-colon at your peril
80 PRINT " "invader$;
90 NEXT
```

The semi-colon at the end of the **PRINT** statement is vital. One important consequence of using **TAG** is that control characters (i.e. those with ASCII codes **0** to **31**) are printed to the screen rather than being obeyed. The cursor moves involved in printing the 'Invader' in the above program become visible if the semi-colon is omitted. We can also see the effect using the 'rocket' as an example.:

```
10 MODE 1
19 REM define 3 characters for rocket
20 SYMBOL 240,0,24,24,24,24,36,36,36
30 SYMBOL 241,36,36,36,36,36,36,36,36
40 SYMBOL 242,66,129,129,129,129,153,195
,129
```

```
50 rocket$=CHR$(240)+CHR$(8)+CHR$(10)+CH
R$(241)+CHR$(8)+CHR$(10)+CHR$(242)
60 TAG
70 graphicsx=300:graphicsy=100
80 FOR y=graphicsy TO 350
89 REM rub out old rocket base
90 MOVE graphicsx,y-24
100 PRINT " ";
109 REM print new rocket (what a mess!)
110 MOVE graphicsx,y
120 PRINT rocket$;
130 NEXT
```

Not quite what we wanted! The problems created by cursor moves for both text and graphics mean it is usually easier to stick to figures that are simple. Figures built up from a series of horizontal characters can be printed after a single **MOVE** statement, because the graphics cursor automatically moves through the width of a single character after printing and is thus correctly positioned to produce the next character. We can see this with the earlier 'lorry' figure, which consisted of a single string:

```
10 MODE 1
19 REM define 3 characters for lorry
20 SYMBOL 240,0,0,96,96,96,127,18,12
21 SYMBOL 241,0,0,0,0,0,255,0,0
22 SYMBOL 242,248,132,132,255,255,255,72
,48
30 lorry$=CHR$(240)+CHR$(241)+CHR$(242)
40 graphicsx=0:graphicsy=200
49 REM join text to graphics cursor
50 TAG
60 FOR x=graphicsx TO 600
70 MOVE x,graphicsy
79 REM print space to rub out back of lo
rry
80 PRINT " "lorry$;
90 NEXT
```

The 'rocket' would have to be printed with a succession of **MOVE** statements, because each character is above the previous ones:

```
10 MODE 1
```

```
19 REM define 3 characters for rocket
20 SYMBOL 240,0,24,24,24,24,36,36,36
30 SYMBOL 241,36,36,36,36,36,36,36,36
40 SYMBOL 242,66,129,129,129,129,153,195
,129
50 rockettop$=CHR$(240)
51 rocketmid$=CHR$(241)
52 rocketbot$=CHR$(242)
60 TAG
70 graphicsx=300:graphicsy=100
80 FOR y=graphicsy TO 350
89 REM rub out old rocket base
90 MOVE graphicsx,y-48
100 PRINT " ";
110 MOVE graphicsx,y
120 PRINT rockettop$;
121 MOVE graphicsx,y-16
122 PRINT rocketmid$;
123 MOVE graphicsx,y-32
124 PRINT rocketbot$;
130 NEXT
```

TAG can be switched off within a program using the **TAGOFF** command. **TAG** is automatically switched off at the end of a program.

Faster movement

You have probably noticed that there is an inevitable price to be paid for this smoother movement of figures: the program runs more slowly. Single character animation is faster, but there are a number of other ways in which we can ensure that the program runs as rapidly as possible.

First, we can speed the program up by using integers (whole numbers) wherever we can. This enables the computer to carry out calculations more quickly. It might seem that we have only used integers in the previous programs — however, the Amstrad treats all numbers as decimals internally unless it is informed otherwise. We can declare particular variables as integers by using the **DEFINT** statement:

```
1 DEFINT g,x,y
```

The Amstrad will now treat any numeric variables beginning with either g, x or y as integers. The difference in speed becomes apparent if we time the same program with and without the use of integers:

```
1 DEFINT g,x,y
10 MODE 1
19 REM define 3 characters for lorry
20 SYMBOL 240,0,0,96,96,96,127,18,12
21 SYMBOL 241,0,0,0,0,0,255,0,0
22 SYMBOL 242,248,132,132,255,255,255,72
,48
30 lorry$=CHR$(240)+CHR$(241)+CHR$(242)
40 graphicsx=0:graphicsy=200
49 REM join text to graphics cursor
50 TAG
55 starttime=TIME
60 FOR x=graphicsx TO 600
70 MOVE x,graphicsy
79 REM print space to rub out back of lo
rry
80 PRINT " "lorry$;
90 NEXT
99 totaltime=TIME
100 TAGOFF
110 LOCATE 1,20
120 PRINT "Time taken was "(totaltime-st
arttime)/300"seconds"
```

A second way of speeding up movement is to take note of the minimum displacement which the Amstrad can successfully display in each mode. There is little point in moving a character horizontally by a single x coordinate in mode 0 because the resolution is so low that printing will take place to exactly the same spot. Move at least four units in mode 0 and two units in mode 1. Vertical resolution is the same in all three modes, but the minimum movement that can be displayed is of 2 units.

Lastly, plan characters so that they have an empty border surrounding them. This ensures that movement in any direction does not leave a trail. The second 'Space Invader' shown in Figure 2.9 will leave lines which need to be erased whenever it is moved. This deletion slows the program down.

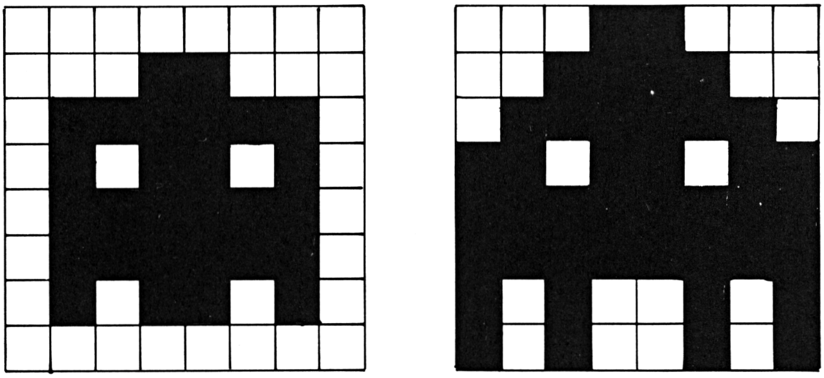


Figure 2.9 Two 'Space Invader' characters: the first is more useful as its border automatically erases the previous image.

What comes next?

Most games programs involve identifying what is present in a nearby screen position — has the racing car hit the wall, and did the laser strike the Space Invader? By using the `TEST (x,y)` command we can discover the `PEN` used at any given graphics position. By carefully choosing the colour of the characters used in a game we can ensure the program behaves differently if we move to a position containing a point of a particular colour. All spiders might be pink, for example, and if we try to move our fly to a spot which proves to be pink the game ends abruptly:

```
1 REM speed things up
5 DEFINT c,x,y
10 MODE 0
19 REM define fly and spider characters
20 SYMBOL 240,0,36,90,90,90,36,0,0
30 SYMBOL 241,145,82,52,31,248,44,74,137
40 fly$=CHR$(240)
50 spider$=CHR$(241)
60 GOSUB 1000
70 xfly=300:yfly=200
80 xnew=xfly:ynew=yfly
90 xtest=xfly:ytest=yfly
100 MOVE xfly,yfly:PLOT xfly,yfly,1
110 GOSUB 2000
120 response$="":flydead=0
```

```
129 REM keep scanning keyboard until the
 fly's dead
130 WHILE response$="" OR flydead=0
140 xnew=xfly:ynew=yfly
150 response$=LOWER$(INKEY$)
159 REM position (xtest,ytest) to colour
-check depends on direction of move
160 IF response$="a" THEN ynew=yfly+2:xt
est=xnew+16:ytest=ynew+8
170 IF response$="z" THEN ynew=yfly-2:xt
est=xnew+16:ytest=ynew-24
180 IF response$="." THEN xnew=xfly+4:xt
est=xnew+48:ytest=ynew-8
190 IF response$="," THEN xnew=xfly-4:xt
est=xnew-16:ytest=ynew-8
200 IF xnew<>xfly OR ynew<>yfly THEN GOS
UB 2000
210 WEND
220 MODE 1
230 END
999 REM set up pink colour
1000 MOVE 0,0
1010 DRAW 0,0,11
1020 TAG
1029 REM draw 10 spiders at random
1030 FOR spiders=1 TO 10
1040 spiderx=INT(600*RND(1)+20)
1050 spidery=INT(300*RND(1)+20)
1060 MOVE spiderx,spidery
1070 PRINT spider$;
1080 NEXT
1090 RETURN
1999 REM test colour at centre of next c
haracter position
2000 colour=TEST(xtest,ytest)
2008 REM if it's pink, the fly's dead
2009 REM not perfect - we can miss the s
pider if the point misses the body!
2010 IF colour=11 THEN flydead=1:SOUND 7
,2000
2019 REM print spider to new position
2020 xfly=xnew:yfly=ynew
2050 MOVE xfly,yfly
2060 PRINT fly$;
2070 RETURN
```

We can easily extend the game by introducing a time element:

```
120 response$="":flydead=0
126 REM this time spider must reach the
top right corner
127 REM as quickly as possible, so set u
p start time
128 oldtime=TIME
129 REM keep scanning keyboard until the
 fly's dead or the corner's reached
130 WHILE (response$="" OR flydead=0) AN
D(xfly<600 OR yfly<300)
140 xnew=xfly:ynew=yfly
150 response$=LOWER$(INKEY$)
159 REM position (xtest,ytest) to colour
-check depends on direction of move
160 IF response$="a" THEN ynew=yfly+2:xt
est=xnew+16:ytest=ynew+8
170 IF response$="z" THEN ynew=yfly-2:xt
est=xnew+16:ytest=ynew-24
180 IF response$="." THEN xnew=xfly+4:xt
est=xnew+48:ytest=ynew-8
190 IF response$="," THEN xnew=xfly-4:xt
est=xnew-16:ytest=ynew-8
200 IF xnew<>xfly OR ynew<>yfly THEN GOS
UB 2000
210 WEND
215 TAGOFF:CLS
220 IF flydead=0 THEN PRINT "Time taken
was: "TIME-oldtime
230 END
```

A useful variation of the `TEST` command is the `TESTR` command which tests the `PEN` present at a position relative to the present position. For example, `TESTR (10,-5)` examines the point which is 10 units to the right of the present point and 5 units down from it. This program demonstrates its use. You must guide the 'car' around the racing track. Don't go off the black 'road'!

```
10 MODE 0
20 GOSUB 1000
30 GOSUB 2000
40 END
1000 PAPER 12
```

```
1010 PEN 0
1020 CLS
1029 REM draw 'track'
1030 sidex=3:sidey=7
1040 FOR y=sidey TO sidey+11
1050 LOCATE sidex,y
1060 PRINT CHR$(143)CHR$(143);
1070 LOCATE sidex+15,y
1080 PRINT CHR$(143)CHR$(143);
1090 NEXT
1100 startx=7:starty=5
1109 FOR count=-1 TO 1
1110 LOCATE startx,starty+count
1120 PRINT STRING$(8,CHR$(143));
1121 LOCATE startx,starty-1
1130 LOCATE startx,starty+15+count
1140 PRINT STRING$(8,CHR$(143));
1141 NEXT
1150 TAG
1160 colour=0
1170 leftx=32:rightx=576
1180 boty=150:topy=330
1190 changey=6:changex=32
1200 PLOT leftx+changex,topy+changey,col
our
1210 FOR count=1 TO 4
1220 ychange=changey*count
1230 xchange=changex*count
1240 MOVE leftx+xchange,topy+ychange
1250 PRINT CHR$(143);
1260 GOSUB 1600
1280 MOVE rightx-xchange,topy+ychange
1290 PRINT CHR$(143);
1300 GOSUB 1600
1320 MOVE leftx+xchange,boty-ychange+8
1330 PRINT CHR$(143);
1340 GOSUB 1600
1360 MOVE rightx-xchange,boty-ychange+8
1370 PRINT CHR$(143);
1380 GOSUB 1600
1400 NEXT
1408 REM 2 car symbols, one for up/down
movement
1409 REM one for left/right movement
1410 SYMBOL 240,0,102,36,126,126,36,102,
```

```
0
1420 SYMBOL 241,0,90,126,24,24,126,90,0
1430 side$=CHR$(240)
1440 up$=CHR$(241)
1450 car$=side$
1460 carx=400:cary=338
1470 PLOT carx+16,cary-4,3
1480 MOVE carx,cary
1490 PRINT car$;
1500 RETURN
1600 FOR count1=1 TO 4
1610 MOVER -32,-16
1620 PRINT CHR$(143);
1630 NEXT
1640 RETURN
1999 REM scan keyboard for key depressio
n
2000 carhit=0
2010 changex=0:changey=0
2020 WHILE carhit=0
2030 response$=LOWER$(INKEY$)
2040 IF response$="a" THEN changex=0:cha
ngey=2:car$=up$:testx=-16:testy=2
2050 IF response$="z" THEN changex=0:cha
ngey=-2:car$=up$:testx=-16:testy=-16
2060 IF response$="." THEN changex=4:cha
ngey=0:car$=side$:testx=0:testy=-8
2070 IF response$="," THEN changex=-4:ch
angey=0:car$=side$:testx=-36:testy=-8
2079 REM Space Bar stops car
2080 IF response$=" " THEN changex=0:cha
ngey=0
2089 REM only draw car again when it has
 been moved
2090 IF changex<>0 OR changey<>0 THEN GO
SUB 3000
2100 WEND
2110 RETURN
2997 REM test colour of pixel next to pr
esent position
2998 REM test colour of pixel next to pr
esent position
2999 REM in next character row/column is
 perfect (but slow)
3000 colourpen=TESTR(testx,testy)
```

```
3009 REM if not INK 0 then car is off th
e track
3010 IF colourpen<>0 THEN carhit=1:SOUND
 7,500
3020 carx=carx+changex:cary=cary+changey
3030 MOVE carx,cary
3040 PRINT car$;
3050 RETURN
```

Exercises

1) Add a few obstacles on the racing track — yellow bales of hay or the burnt-out remains of a car.
2) Create a multiple-character green caterpillar that ambles slowly across the screen.
3) Add some red berries which are printed at random positions on-screen. If the caterpillar eats a berry, it turns blue and dies.
4) Add keyboard controls so that you can attempt to guide the caterpillar on its perilous journey by moving it vertically so that it avoids the berries. The creature is safe if it reaches a rich swathe of green grass on the right of the screen.

Chapter Three

Graphs and charts

In the last chapter we looked at the lighter use of graphics, in games, but there are much more serious applications, even on a microcomputer. The last few years have seen a proliferation of sophisticated software suitable for small businesses, and many of these programs take as their aim the presentation of information in a more easily understood manner. Rather than providing endless lists of facts and figures, the software manipulates the data and from it produces graphs, bar charts, pie charts, or a combination of all three. The use of colour and high-resolution graphics makes it easy to display trends or highlight particular features. The computer has the added advantage in that it can rapidly recalculate and display a new graph or chart to illustrate the consequences of, for example, a drop in sales revenue.

We shall concentrate in this chapter on software to produce the three most familiar forms of data presentation: graphs, bar charts and pie charts. At this stage in the book it is important to be aware of some of the rules-of-thumb that should be used when planning software.

First, it is unwise to attempt to write a program as a whole. It is much easier to develop, debug and amend a program if it is written in MODULES, short sections of the program that have a specific purpose, i.e. to draw the axes of a graph, or colour a bar on a bar chart.

Second, a program is very inflexible if it is tied to specific values. A program to draw the axes for a particular graph might work very well. But if it contains lines like:

```
100 MOVE 308,390
110 DRAW 308,120
120 DRAW 630,120
```

it becomes difficult to use again, and producing a new graph may well involve rewriting the program. It is far better to use VARIABLES on all possible occasions. This has many advantages: variable names are more meaningful than strings of numbers in **MOVE** and **DRAW** commands, and the program is easier to understand and modify if we return to it after many months and wish to change it.

Additionally, the use of variables means that we can write a general purpose program that will produce a graph for any set of data — the only changes necessary will be to the data itself. There will be no need to edit program line after program line to take account of the new circumstances.

Writing a program of this form involves more thought initially, and the software may take longer to develop. It is well worth the extra trouble. By adopting this approach we avoid writing numerous programs to carry out basically the same tasks.

Point and line graphs

The first question that arises when drawing graphs is that of the resolution required. On the Amstrad we have a choice of modes, each with differing horizontal resolutions and the same vertical resolution. In general it is better to draw a graph with many points in mode 2, to take advantage of the high resolution. Unfortunately we are limited to only two colours in mode 2. If the number of points to be plotted is fewer than 300 mode 1 gives a reasonable compromise between the demands of resolution and colour: it offers 320 individually addressable points horizontally, along with a choice of four colours.

The Amstrad has been designed so that a change of mode does not affect the range of the graphics coordinates. The program that follows will thus run equally well in any of the modes, although clearly the resolution will vary.

We will begin by developing a (very brief!) program. For the moment the graph will be drawn using the entire screen and we will leave the problem of labelling until later. We will write the program as a series of subroutines. This gives us more flexibility, and makes it possible to plot several sets of data on a single graph, or draw multiple graphs, without any radical alterations to the program:

```
10 MODE 1
20 GOSUB 500
30 GOSUB 800
40 END
499 REM draw axes
500 ypoints=399
510 xpoints=639
520 MOVE 0,ypoints
530 DRAW 0,0,1
540 DRAW xpoints,0
550 minx=200
560 maxx=4000
570 diffx=maxx-minx
580 miny=100
590 maxy=1000
600 diffy=maxy-miny
610 pointx=diffx/xpoints
620 pointy=diffy/ypoints
690 RETURN
799 REM read points from data and plot t
heir position
800 READ noofpoints
805 DIM x(noofpoints),y(noofpoints)
810 FOR count=1 TO noofpoints
815 READ x(count):xdispl=(x(count)-minx)
/pointx
820 READ y(count):ydispl=(y(count)-miny)
/pointy
825 PLOT xdispl,ydispl
830 NEXT
990 RETURN
1000 DATA 5,200,100,1000,200,1500,300,25
00,600,4000,1000
```

Line **520** draws the axes of the graph. The origin is placed at the bottom left corner of the screen. It is essential that we know the minimum and maximum values of the data, so that the graph can be scaled to ensure that all the points are on-screen. These values are stated explicitly in lines **550** to **600**, although we shall see later that the computer can itself derive this information from the data provided. Once the Amstrad has found the difference between the minimum and maximum values, it can calculate how much each unit along the x and y axis will have to represent for all the data to fit on, lines **610** and 620. The origin will represent the point (min x, min y) and the top right corner of the screen will be (max x, max y).

Finally, the program reads in the x and y coordinates for the data, scales the points and plots them, lines 800 to 830. The data in this case is already ordered from the lowest to the highest x coordinate. Randomly ordered data will be dealt with at a later stage.

You may care to run the program a few times with changed values for max x and max y to see the effect it has. Doubling max x will 'squash' the graph towards the left, as the program leaves room for higher values of x that might be present in the data. Doubling max y 'squashes' the graph downwards. The variables min x and min y give similar effects when changed. If you dislike the fact that the corner of the graph is (100,200) and not (0,0), change min x and min y to 0. (Note that this wastes part of the screen area as there are no points displayed here.) Remember that the data at line 1000 falls within a particular range — if you change the values of min x etc. too radically you will lose some points from the graph!

Running the program reveals a few problems — the graph and points are rather difficult to see. A few modifications can improve the situation:

```
10 MODE 1
20 GOSUB 500
30 GOSUB 800
40 END
499 REM draw axes
500 ypoints=399
510 xpoints=639
520 MOVE 0,ypoints
530 DRAW 0,0,1
540 DRAW xpoints,0
550 minx=200
560 maxx=4000
570 diffx=maxx-minx
580 miny=100
590 maxy=1000
600 diffy=maxy-miny
610 pointx=diffx/xpoints
620 pointy=diffy/ypoints
690 RETURN
```

```
799 REM read points from data and plot t
heir position
800 READ noofpoints
805 DIM x(noofpoints),y(noofpoints)
806 READ pencolour,papercolour
807 INK 0,papercolour
808 INK 1,pencolour
809 PAPER 0:PEN 1
810 FOR count=1 TO noofpoints
815 READ x(count):xdispl=(x(count)-xmin)
/pointx
820 READ y(count):ydispl=(y(count)-ymin)
/pointy
825 PLOT xdispl,ydispl
830 NEXT
990 RETURN
1000 DATA 5,0,24,200,100,1000,200,1500,3
00,2500,600,4000,1000
```

The colours used to draw the graph are now specified in the **DATA** statement. We could go even further and specify the **MODE** here, but let's stick to mode 1! The points can be made more visible by plotting a cross or square at each position. This is easily done using relative moves:

```
10 MODE 1
20 GOSUB 500
30 GOSUB 800
40 END
499 REM draw axes
500 ypoints=399
510 xpoints=639
520 MOVE 0,ypoints
530 DRAW 0,0,1
540 DRAW xpoints,0
550 minx=200
560 maxx=4000
570 diffx=maxx-minx
580 miny=100
590 maxy=1000
600 diffy=maxy-miny
610 pointx=diffx/xpoints
620 pointy=diffy/ypoints
690 RETURN
799 REM read points from data and plot t
```

```
heir position
800 READ noofpoints
805 DIM x(noofpoints),y(noofpoints)
806 READ pencolour,papercolour
807 INK 0,papercolour
808 INK 1,pencolour
809 PAPER 0:PEN 1
810 crossx=10
811 crossy=10
814 FOR count=1 TO noofpoints
815 READ x(count):xdispl=(x(count)-minx)
/pointx
820 READ y(count):ydispl=(y(count)-miny)
/pointy
825 PLOT xdispl,ydispl
830 MOVER -crossx,crossy
840 DRAWR 2*crossx,-2*crossy
850 MOVER -2*crossx,0
860 DRAWR 2*crossx,2*crossy
870 MOVER -crossx,-crossy
880 NEXT
990 RETURN
1000 DATA 5,0,24,200,100,1000,200,1500,3
00,2500,600,4000,1000
```

The movement to create one 'arm' of the cross is given as two variables rather than in the data because the size of such a marker is likely to remain fixed from run to run of the program. You may care to make the cross smaller or larger to your own taste: you need only change two lines.

The program will be more useful if there is an option to join the points. This can be indicated by setting a variable 'flag' to one of two values to show whether the points are to be plotted separately or joined:

```
10 MODE 1
20 GOSUB 500
30 GOSUB 800
40 END
499 REM draw axes
500 ypoints=399
510 xpoints=639
520 MOVE 0,ypoints
530 DRAW 0,0,1
540 DRAW xpoints,0
```

```
550 minx=200
560 maxx=4000
570 diffx=maxx-minx
580 miny=100
590 maxy=1000
600 diffy=maxy-miny
610 pointx=diffx/xpoints
620 pointy=diffy/ypoints
690 RETURN
799 REM read points from data and plot t
heir position
800 READ noofpoints
805 DIM x(noofpoints),y(noofpoints)
806 READ pencolour,papercolour
807 INK 0,papercolour
808 INK 1,pencolour
809 PAPER 0:PEN 1
810 crossx=10
811 crossy=10
812 flag=1
814 FOR count=1 TO noofpoints
815 READ x(count):xdispl=(x(count)-minx)
/pointx
816 x(count)=xdispl
820 READ y(count):ydispl=(y(count)-miny)
/pointy
821 y(count)=ydispl
825 PLOT xdispl,ydispl
830 MOVER -crossx,crossy
840 DRAWR 2*crossx,-2*crossy
850 MOVER -2*crossx,0
860 DRAWR 2*crossx,2*crossy
870 MOVER -crossx,-crossy
874 REM if flag set connect point to pre
vious one
875 IF flag=1 AND count>1 THEN DRAW x(co
unt-1),y(count-1)
880 NEXT
990 RETURN
1000 DATA 5,0,24,200,100,1000,200,1500,3
00,2500,600,4000,1000
```

The major deficiency in the program is the lack of labelling. As the graph is 'tied' to the bottom left corner of the screen there is no room for labelling, and it seems as if the program will require drastic modification. In fact the use of variables and the Amstrad's ability to alter the graphics origin make it remarkably easy to move the axes:

```
10 MODE 1
20 GOSUB 500
30 GOSUB 800
40 END
499 REM draw axes
500 ox=100:oy=50
505 ORIGIN ox,oy
510 ypoints=399-oy
515 xpoints=639-ox
520 MOVE 0,ypoints
530 DRAW 0,0,1
540 DRAW xpoints,0
550 minx=200
560 maxx=4000
570 diffx=maxx-minx
580 miny=100
590 maxy=1000
600 diffy=maxy-miny
610 pointx=diffx/xpoints
620 pointy=diffy/ypoints
690 RETURN
799 REM read points from data and plot t
heir position
800 READ noofpoints
805 DIM x(noofpoints),y(noofpoints)
806 READ pencolour,papercolour
807 INK 0,papercolour
808 INK 1,pencolour
809 PAPER 0:PEN 1
810 crossx=10
811 crossy=10
812 flag=1
814 FOR count=1 TO noofpoints
815 READ x(count):xdispl=(x(count)-minx)
/pointx
816 x(count)=xdispl
820 READ y(count):ydispl=(y(count)-miny)
/pointy
821 y(count)=ydispl
825 PLOT xdispl,ydispl
830 MOVER -crossx,crossy
```

```
840 DRAWR 2*crossx,-2*crossy
850 MOVER -2*crossx,0
860 DRAWR 2*crossx,2*crossy
870 MOVER -crossx,-crossy
874 REM if flag set connect point to pre
vious one
875 IF flag=1 AND count>1 THEN DRAW x(co
unt-1),y(count-1)
880 NEXT
990 RETURN
1000 DATA 5,0,24,200,100,1000,200,1500,3
00,2500,600,4000,1000
```

Lines **510** and **515** must be modified because we are no longer using the entire screen for the graph. The remainder of the program can remain exactly as it is: all lines drawn and points plotted will be relative to the new origin, as you can see if you run the program. Give ox and oy new values and you can see that the shape of the graph remains the same no matter where you put the origin, although its size will obviously vary.

We now have room to mark intervals on the axes and label them. This is not something which can be completely automated. Labelling the y axis should pose no problem — hopefully we have chosen an origin which leaves sufficient room for the labels!

Clearly, if we have room on the x axis for 10 marks we could instead choose to use a smaller number, such as 5. The maximum number of intervals marked is limited by two factors: the resolution of the mode, and the width of the characters to be printed beneath each mark. The computer can calculate whether a suggested interval will result in printed characters overlapping, but it cannot really select a reasonable interval less than this for itself. Humans have a taste for graph intervals of 0.5, 10, 20, 25, 100, etc., depending on the circumstances. We shall leave selection of this interval as a human function: the Amstrad will reject unreasonable values.

The labelling can be broken into two parts: marking the intervals with 'ticks', and printing and characters. The first action cannot proceed if the subsequent printing will overlap. We must inform the computer of the maximum length of the strings which will be printed so that it can calculate whether the strings can be successfully fitted in beneath the x axis:

```
503 REM hide graph until labelling compl
ete
504 INK 0,24:INK 1,24
628 REM number of 'ticks' to be marked o
n x and y axes
629 REM note this includes a 'tick' at t
he origin
630 xrange=5
640 yrange=10
649 REM graphics distance between each '
tick'
650 xwidth=INT(xpoints/xrange)
660 yheight=INT(ypoints/yrange)
668 REM maximum character length of numb
ers to be printed on x axis
669 REM you may have to change this for
your own data
670 maxxstring=4
673 REM size of character in terms of gr
aphics points for mode 1
674 REM change charwidth to 32 if progra
m is run in mode 0, 8 if mode 2
675 charwidth=16
676 charheight=16
679 REM calculate max string length in t
erms of graphics points
680 graphxstring=charwidth*maxxstring
688 REM do not label axes if numbers are
 too wide to fit
689 REM or if distance between 'ticks' n
arrower than 1 character
690 IF xwidth<graphxstring OR xwidth<cha
rwidth THEN RETURN
691 REM ditto for y axis
692 maxystring=4
694 graphystring=charwidth*maxystring
696 IF ox<graphystring OR yheight<charhe
ight THEN RETURN
699 REM length of 'ticks' when drawn on
axes
700 xtick=6
702 ytick=8
703 REM each 'tick' is labelled xvalue h
igher than the previous one
```

```
704 xvalue=diffx/xrange
706 TAG
707 REM do it xrange times to give the r
equired number of 'ticks'
708 FOR count=0 TO xrange
710 MOVE 0,0
711 REM move along x axis to start of 't
ick'
712 MOVER xwidth*count,0
713 REM draw tick
714 DRAWR 0,-xtick
728 NEXT
729 REM ditto for y axis
730 yvalue=diffy/yrange
732 FOR count=0 TO yrange
734 MOVE 0,0
736 MOVER 0,yheight*count
738 DRAWR -ytick,0
754 NEXT
790 RETURN
```

As with the printing of the crosses at points on the graph, you may prefer your 'ticks' to be more or less obtrusive. Their size can be changed by amending the appropriate lines.

Rather than carry out a recalculation of the tick positions when printing the x and y values, we can slot the printing routine in at the appropriate point when the ticks are drawn:

```
708 FOR count=0 TO xrange
710 MOVE 0,0
711 REM move along x axis to start of 't
ick'
712 MOVER xwidth*count,0
713 REM draw tick
714 DRAWR 0,-xtick
715 REM get to correct position to print
 number relative to 'tick'
716 MOVER -charwidth/2,-xtick
718 number$=STR$(minx+count*xvalue)
719 REM truncate value if it's too long
to fit
720 number$=MID$(number$,2,maxxstring)
722 length=LEN(number$)
723 REM strip off decimal point if it en
```

```
ds in one
724 IF MID$(number$,length)="." THEN num
ber$=MID$(number$,1,length-1)
726 PRINT number$;
728 NEXT
729 REM ditto for y axis
730 yvalue=diffy/yrange
732 FOR count=0 TO yrange
734 MOVE 0,0
736 MOVER 0,yheight*count
738 DRAWR -ytick,0
740 MOVER -graphystring,charheight/2
742 number$=STR$(miny+count*yvalue)
744 number$=MID$(number$,2,maxystring)
746 length=LEN(number$)
748 IF MID$(number$,length)="." THEN num
ber$=MID$(number$,1,length-1)
749 REM pad short numbers out so they al
l touch a 'tick'
750 IF LEN(number$)<maxystring THEN numb
er$=STRING$(maxystring-LEN(number$)," ")
+number$
752 PRINT number$;
754 NEXT
790 RETURN
```

We would also want to label the axes and give the overall graph a title. Again we must assume that the origin has been sensibly chosen so that there is room for the characters!

```
755 REM labels for axes
756 xlabel$="Mice population"
758 ylabel$="Cheese eaten in gm"
759 REM find label length in terms of gr
aphic points
760 xlablength=LEN(xlabel$)*charwidth
761 REM find space to leave before print
ing label, to ensure it is centred
762 xlabstart=(xpoints-xlablength)/2
763 REM move down 2 characters from x ax
is to print label
764 MOVE xlabstart,-2*charheight
766 PRINT xlabel$;
767 REM ditto for y axis label
768 ylablength=LEN(ylabel$)*charheight
```

```
770 ylabstart=(ypoints+ylablength)/2
771 REM each character must be printed s
eparately as this is a vertical label
772 FOR count=1 TO LEN(ylabel$)
773 REM extract character from label
774 char$=MID$(ylabel$,count,1)
775 REM move left 2 characters from y ax
is numbers to print character
776 MOVE -charwidth*(maxystring+2),ylabs
tart-charheight*(count-1)
778 PRINT char$;
780 NEXT
790 RETURN
```

The program is not complete, because it will only work on previously ordered data. This is adequate for many purposes, e.g. measurement of rainfall over a period of time or fluctuations in the state of your bank account over the months (though you might need a negative axis here!). If the data values are to be sorted into ascending order it will no longer be adequate just to read the values, scale them and plot them immediately. Each value will have to be stored so that the computer can compare all the values, and, if necessary, re-order them. All the data is read into an array and then sorted:

```
814 FOR count=1 TO noofpoints
815 READ x(count):xdispl=(x(count)-minx)
/pointx
816 x(count)=xdispl
820 READ y(count):ydispl=(y(count)-miny)
/pointy
821 y(count)=ydispl
825 NEXT
829 REM take each x array value in turn
830 FOR count=2 TO noofpoints
840 FOR value=count TO 2 STEP -1
850 IF x(value)<x(value-1) THEN GOSUB 15
00 ELSE value=2
860 NEXT
870 NEXT
1499 REM swap x,y array values around so
 that lowest x coordinate comes first
1500 tempx=x(value):tempy=y(value)
```

```
1510 x(value)=x(value-1):y(value)=y(valu
e-1)
1520 x(value-1)=tempx:y(value-1)=tempy
1530 RETURN
```

A few other lines must be modified as the coordinates are no longer read directly from data and plotted.
The sort used here is known as an INSERTION SORT. The first two x coordinates are ordered and the third coordinate then inserted into its correct position with regard to the first two. The process is then repeated with the insertion of the fourth coordinate and so on until the coordinates are all ordered from lowest to highest.

Sorting of numeric and string variables is a topic in its own right within computing. If you are sorting several hundred numbers you may find the above process too slow, and you may prefer to use another sort. For example the bubble sort is particularly useful when the data is already partially ordered. Ultimately, really rapid sorting of a large set of data can only be achieved using a machine code routine.

One advantage of sorting the data into x coordinate order is that it gives us an opportunity of letting the computer calculate for itself the values of min x, and min y, max x and max y:

```
880 FOR count=1 TO noofpoints
890 PLOT x(count),y(count)
900 MOVER -crossx,crossy
910 DRAWR 2*crossx,-2*crossy
920 MOVER -2*crossx,0
930 DRAWR 2*crossx,2*crossy
940 MOVER -crossx,-crossy
950 REM if flag set connect point to pre
vious one
960 IF flag=1 AND count>1 THEN DRAW x(co
unt-1),y(count-1)
970 NEXT
990 RETURN
999 REM this time data is disordered
1000 DATA 5,0,24,1000,200,4000,1000,200,
100,2500,600,1500,300
```

The program still has a few weaknesses, which you can attempt to rectify in the exercise following, but it is adequate for most purposes.

Exercises

1) What changes need to be made if the program is to run successfully in modes 0 or 2? What values which vary with the mode could usefully be replaced by variables?
2) At present the y axis is always drawn on the left of the graph, and the x axis at its base. Modify the program so that when the data includes some positive coordinates and some negative ones the x and/or y axes are drawn through (0,0).
3) Modify the program so that successive sets of data within the same range can be displayed on the same graph in different colours. Don't repeat sections of the program to draw the other lines: identify the parts of the program you need to use and call them as subroutines.
4) Modify the program so that two graphs with different scales are displayed on the upper and lower halves of the screen. (You will need to amend the value of the variable y points and change the origin twice.)
5) Extend the program so that the data for the graph can be read in or saved to a file.

Bar charts

Drawing a bar chart is much easier now that we have a program to draw a graph, because we have solved most of the problems in the previous section. The process is simplified further because we are only dealing with the y coordinates: once we know the number of data elements it is easy to calculate the width of each bar. Additionally the data is already ordered, so there is no reason to sort it. The similarities to the previous program point up the advantages of extensive use of variables and subroutines:

```
10 MODE 1
20 GOSUB 500
30 GOSUB 900
40 END
499 REM draw axes
500 ox=100:oy=50
503 REM hide graph until labelling compl
ete
```

```
504 INK 0,24:INK 1,24
505 ORIGIN ox,oy
510 ypoints=399-oy
515 xpoints=639-ox
520 MOVE 0,ypoints
530 DRAW 0,0,1
540 DRAW xpoints,0
580 miny=100
590 maxy=1000
600 diffy=maxy-miny
620 pointy=diffy/ypoints
628 REM number of 'ticks' to be marked o
n  y axis only this time
629 REM xrange gives number of bars
630 READ noofbars:xrange=noofbars
640 yrange=9
649 REM graphics distance between each
tick'
650 xwidth=INT(xpoints/xrange)
660 yheight=INT(ypoints/yrange)
673 REM size of character in terms of gr
aphics points for mode 1
674 REM change charwidth to 32 if progra
m is run in mode 0, 8 if mode 2
675 charwidth=16
676 charheight=16
692 maxystring=4
694 graphystring=charwidth*maxystring
696 IF ox<graphystring OR yheight<charhe
ight THEN RETURN
699 REM length of 'ticks' when drawn on
axes
700 xtick=6
702 ytick=8
703 REM each 'tick' is labelled xvalue h
igher than the previous one
706 TAG
707 REM do it xrange times to give the r
equired number of 'ticks'
730 yvalue=diffy/yrange
732 FOR count=0 TO yrange
734 MOVE 0,0
736 MOVER 0,yheight*count
738 DRAWR -ytick,0
```

```
740 MOVER -graphystring,charheight/2
742 number$=STR$(miny+count*yvalue)
744 number$=MID$(number$,2,maxystring)
746 length=LEN(number$)
748 IF MID$(number$,length)="." THEN num
ber$=MID$(number$,1,length-1)
749 REM pad short numbers out so they al
l touch a 'tick'
750 IF LEN(number$)<maxystring THEN numb
er$=STRING$(maxystring-LEN(number$)," ")
+number$
752 PRINT number$;
754 NEXT
755 REM labels for axes
756 xlabel$="Mice population"
758 ylabel$="Cheese eaten in gm"
759 REM find label length in terms of gr
aphic points
760 xlablength=LEN(xlabel$)*charwidth
761 REM find space to leave before print
ing label, to ensure it is centred
762 xlabstart=(xpoints-xlablength)/2
763 REM move down 2 characters from x ax
is to print label
764 MOVE xlabstart,-2*charheight
766 PRINT xlabel$;
767 REM ditto for y axis label
768 ylablength=LEN(ylabel$)*charheight
770 ylabstart=(ypoints+ylablength)/2
771 REM each character must be printed s
eparately as this is a vertical label
772 FOR count=1 TO LEN(ylabel$)
773 REM extract character from label
774 char$=MID$(ylabel$,count,1)
775 REM move left 2 characters from y ax
is numbers to print character
776 MOVE -charwidth*(maxystring+2),ylabs
tart-charheight*(count-1)
778 PRINT char$;
780 NEXT
790 RETURN
799 REM read points from data and plot t
heir position
800 DIM y(noofbars)
```

```
806 READ pencolour,papercolour
807 INK 0,papercolour
808 INK 1,pencolour
809 PAPER 0:PEN 1
814 FOR count=1 TO noofbars
815 READ y
819 REM height of bar scaled according t
o position of origin
820 y(count)=(y-miny)/pointy
825 NEXT
829 REM reduce bar width to leave space
between bars
830 barwidth=xwidth-4
840 FOR count=1 TO noofbars
850 MOVE 0,0
860 DRAWR 0,y(count),1
870 DRAWR barwidth,0
880 DRAWR 0,-y(count)
890 ox=ox+xwidth
899 REM shift origin ready to draw next
bar
900 ORIGIN ox,oy
910 NEXT
990 RETURN
999 REM height of bars only required as
data this time - width is fixed
1000 DATA 10,0,24,350,190,760,440,990,12
4,846,545,666,222
```

Subroutine **800** draws each bar as a series of relative moves with respect to the origin. By shifting the origin a fixed distance between the drawing of each bar we can use a loop to draw a succession of bars.

The final display is more impressive if the bars are shaded different colours. Unfortunately the Amstrad has no command to fill a graphics area with colour, so the bars must be shaded by drawing single lines as rapidly as possible. In the last chapter we identified some ways of speeding a program up, and we can usefully apply some of that knowledge here:

```
830 barwidth=xwidth-4
835 colour=2
840 FOR count=1 TO noofbars
844 REM colour alternate bars differentl
y
```

```
845 IF colour=3 THEN colour=2 ELSE colou
r=3
848 REM bars can be filled more rapidly
taking resolution into account
849 REM so use a STEP based on the chara
cter width for the mode
850 FOR bar=0 TO barwidth STEP charwidth
/8
860 MOVE 0+bar,0
870 DRAWR 0,y(count),colour
880 NEXT
890 ox=ox+xwidth
899 REM shift origin ready to draw next
bar
900 ORIGIN ox,oy
910 NEXT
990 RETURN
```

The lines are drawn vertically rather than horizontally because most bars will be higher than they are wide, and this means it takes fewer vertical lines to shade them in. Really rapid colour-fill is only possible with machine code routines.

An interesting variation on the bar chart is to draw 'solid' bars to give a three-dimensional appearance:

```
830 barwidth=xwidth-4
831 REM barside is the x width of the si
de of the bar
832 REM bartop is the y height of the ba
ck of the bar above the front
833 REM put values of your own to get de
eper or shallower bars
834 barside=barwidth/4:bartop=barside
835 colour=2
840 FOR count=1 TO noofbars
844 REM colour alternate bars differentl
y
845 IF colour=3 THEN colour=2 ELSE colou
r=3
846 bartopcount=0
848 REM this time we must fill in the ba
r side as well
849 REM so the width of a bar is barwidt
h+barside
850 FOR bar=0 TO barwidth+barside STEP c
```

```
harwidth/8
859 REM draw end of line in another colo
ur - helps outline the bar
860 PLOT 0+bar,0,1
869 REM draw line up to highest point on
 bar at this x coordinate
870 DRAWR 0,y(count)+bartopcount,colour
871 REM draw other end of line in a diff
erent colour
872 PLOTR 0,0,1
873 REM next line must be drawn a little
 higher
874 REM and so on until the height of th
e bar back is reached
875 bartopcount=bartopcount+charwidth/8:
IF bartopcount>bartop THEN bartopcount=b
artop
880 NEXT
881 REM now draw outline of bar in anoth
er colour
882 y=y(count):MOVE 0,0
883 DRAWR 0,y,1
884 DRAWR barwidth,0
885 DRAWR 0,-y
886 MOVER 0,y
887 DRAWR barside,bartop
888 DRAWR 0,-y-bartop
890 ox=ox+xwidth
899 REM shift origin ready to draw next
bar
900 ORIGIN ox,oy
910 NEXT
990 RETURN
```

Exercises

1) Make changes to the program so that it runs correctly in mode 0 or 2.
2) Modify the program so that it draws a horizontal bar chart rather than a vertical one.
3) Extend the 3D bar chart so that a succession of bars may be drawn, each set 'in front of' the previous set.

Pie charts

A pie chart has little in common with graphs or bar charts, and here we must develop a completely new program although again we will use subroutines. The resolution is important for the pie chart, as it involves the accurate drawing of a circle, but the use of colour gives a pie chart more impact, so we shall again compromise and use mode 1.

The program to produce our first pie chart simply draws a circle and divides it into sectors of the appropriate size, each with a coloured outline:

```
10 MODE 1
20 GOSUB 1000
30 GOSUB 2000
40 GOSUB 3000
50 END
999 REM read from data circle centre coo
rdinates plus radius
1000 READ centrex,centrey
1010 READ radius
1020 READ numberofvalues
1030 DIM value(numberofvalues),angle(num
berofvalues)
1039 REM add values together so circle c
an be divided appropriately
1040 totalofvalues=0
1050 FOR count=1 TO numberofvalues
1060 READ value(count)
1070 totalofvalues=totalofvalues+value(c
ount)
1080 NEXT
1090 RETURN
1100 DATA 200,200,120,4,1,2,3,4
1999 REM calculate angle for each sector
2000 FOR count=1 TO numberofvalues
2010 angle(count)=2*PI*value(count)/tota
lofvalues
2020 NEXT
2030 RETURN
3000 REM we'll use this soon!
3010 startangle=0
3020 stepsize=PI/60
3030 colour=1
```

```
3039 REM in mode 0 change noofcolours to
 15 (background excluded)
3040 noofcolours=3
3050 FOR count=1 TO numberofvalues
3059 REM calculate end angle for sector
3060 endangle=startangle+angle(count)
3069 REM change colour for each sector
3070 colour=1+(colour+1)MOD noofcolours
3079 REM make sure first and last sector
s are different colours
3080 IF count=numberofvalues AND numbero
fvalues MOD noofcolours=1 THEN colour=1+
(colour+1)MOD noofcolours
3090 MOVE centrex,centrey:DRAW centrex+r
adius*SIN(startangle),centrey+radius*COS
(startangle),colour
3099 REM draw sector
3100 FOR angle=startangle TO endangle ST
EP stepsize
3110 DRAW centrex+radius*SIN(angle),cent
rey+radius*COS(angle)
3120 NEXT
3129 REM update start angle for next sec
tor
3130 startangle=endangle
3140 NEXT
3150 RETURN
```

Subroutine **1000** reads the values from **DATA**, and finds their sum. This is needed to find the angle of the sector representing that particular value in the pie chart, subroutine **2000**. Each sector is outlined in a different colour, subroutine **3000**.

Running the program reveals that for a pie chart with a large radius the chart is drawn rather slowly. We can speed things up by making the centre of the circle the new origin: this makes the calculation more rapid:

```
3000 ORIGIN centrex,centrey
3090 MOVE 0,0:DRAW radius*SIN(startangle
),radius*COS(startangle),colour
3099 REM draw sector
3100 FOR angle=startangle TO endangle ST
EP stepsize
3110 DRAW radius*SIN(angle),radius*COS(a
ngle)
```

```
3120 NEXT
```

The need to calculate the sine and cosine of angles slows things down, but this is one occasion when we really can't use integers instead! However, just to demonstrate that there is more than one way to crack an egg (or draw a pie chart), here is a much faster program. This only calculates a single sine and cosine, and then uses these values to determine the next point on the circumference:

```
2000 radval=radius*4
2005 FOR count=1 TO numberofvalues
2010 angle(count)=radval*value(count)/to
talofvalues
2020 NEXT
2030 RETURN
3000 ORIGIN centrex,centrey
3009 REM loops use only integers with th
is method - speeds it up
3010 DEFINT c
3030 colour=1
3035 thesin=SIN(2*PI/radval):thecos=COS(
2*PI/radval)
3037 x1=radius:y1=0
3039 REM in mode 0 change noofcolours to
 15 (background excluded)
3040 noofcolours=3
3050 FOR count=1 TO numberofvalues
3069 REM change colour for each sector
3070 colour=1+(colour+1)MOD noofcolours
3079 REM make sure first and last sector
s are different colours
3080 IF count=numberofvalues AND numbero
fvalues MOD noofcolours=1 THEN colour=1+
(colour+1)MOD noofcolours
3099 REM draw sector
3100 FOR count1=1 TO angle(count)
3110 x=x1*thecos-y1*thesin
3112 y=x1*thesin+y1*thecos
3114 PLOT x,y,colour
3116 x1=x:y1=y
3120 NEXT
3129 REM draw line to centre for this se
ctor
3130 MOVE 0,0:DRAW x,y
```

```
3140 NEXT
3150 RETURN
```

The pie chart is more impressive if each sector is coloured, and the first program is easily modified to do this. The simplest method is to draw lines successively from the centre to each point on the circumference:

```
1 REM based on progam before last
2 REM change only these lines:
3020 stepsize=PI/500
3109 REM this time draw from centre to c
ircumference
3110 MOVE 0,0:DRAW radius*SIN(angle),rad
ius*COS(angle)
```

This runs even more slowly, but unfortunately unless extremely small steps are taken some pixels within a sector are not touched by the lines, and remain in the background colour. One way of speeding things up is to colour-fill only alternate sectors, leaving some sectors in the background colour. This approach can result in further time improvement if the values are read into an array and then sorted into ascending order. By judicious rearrangement of the order in which the sectors are plotted, we can ensure that only the smaller sectors within the circle are colour-filled. However, this rather defeats the object of using mode 1, and the labelling of the sectors is vital, otherwise the pie-chart can be very confusing!

It is possible to speed the program up still further by calculating the coordinates of the circumference and storing them in an array, and using the array values when the pie-chart is drawn. This is not an approach to take if you are short of memory, because the arrays required are very large. This is easier to do with the alternative pie-chart program:

```
10 MODE 1
20 GOSUB 1000
30 GOSUB 2000
40 GOSUB 3000
50 END
999 REM can use integers in many places
here - speeds it up
1000 DEFINT a,c,n,r,v
1005 READ centrex,centrey
```

```
1010 READ radius
1020 READ numberofvalues
1030 DIM value(numberofvalues),angle(num
berofvalues)
1039 REM add values together so circle c
an be divided appropriately
1040 totalofvalues=0
1050 FOR count=1 TO numberofvalues
1060 READ value(count)
1070 totalofvalues=totalofvalues+value(c
ount)
1080 NEXT
1090 RETURN
1100 DATA 200,200,120,4,1,2,3,4
1998 REM radval effects speed of drawing
  and extent of shading
1999 REM try radius*3 for a faster drawi
ng with a few pixels unshaded
2000 radval=radius*10
2001 totangle=0
2005 FOR count=1 TO numberofvalues
2010 angle(count)=radval*value(count)/to
talofvalues
2015 totangle=totangle+angle(count)
2020 NEXT
2030 RETURN
3000 ORIGIN centrex,centrey
3010 DEFINT c
3020 countangle=0
3030 colour=1
3033 REM only need to calculate one sin
and cos this method
3034 REM makes it faster
3035 thesin=SIN(2*PI/radval):thecos=COS(
2*PI/radval)
3036 x1=radius:y1=0
3037 REM calaculate coordinates of circu
mference points
3038 GOSUB 4000
3039 REM in mode 0 change noofcolours to
 15 (background excluded)
3040 noofcolours=3
3050 FOR count=1 TO numberofvalues
3069 REM change colour for each sector
```

```
3070 colour=1+(colour+1)MOD noofcolours
3079 REM make sure first and last sector
s are different colours
3080 IF count=numberofvalues AND numbero
fvalues MOD noofcolours=1 THEN colour=1+
(colour+1)MOD noofcolours
3099 REM draw sector
3100 FOR count1=countangle TO countangle
+angle(count)
3114 MOVE 0,0:DRAW x(count1),y(count1),c
olour
3120 NEXT
3124 REM update start position for next
sector
3125 countangle=countangle+angle(count)
3140 NEXT
3150 RETURN
3997 REM normally this calculation would
 be carried out
3998 REM at some convenient point in a l
onger program
3999 REM when the delay due to calculati
on would not be obvious
4000 DIM x(totangle),y(totangle)
4010 FOR count=1 TO totangle
4020 x=x1*thecos-y1*thesin
4030 y=x1*thesin+y1*thecos
4040 x(count)=x
4050 y(count)=y
4060 x1=x:y1=y
4070 NEXT
4080 RETURN
```

The above programs demonstrate the difficulty of ensuring that an area is completely filled with a colour. This only becomes certain if we use a colour-fill method which plots individual points, as we shall see in the next chapter.

Exercises

1) Extend the pie-chart program so that each sector is suitably labelled.
2) Modify the program so that several pie-charts with the

same radius are drawn on-screen. (This is definitely an occasion to use arrays as it avoids the need to recalculate the circumference points for each circle.)

3) Write a program that superimposes successively smaller pie-charts on top of each other, with the sectors being colour-filled.

Patterns and pictures

In Chapter 1 we saw how easy it is to produce patterns just by using a combination of MOVE and DRAW commands:

```
10 MODE 1
20 x=320:y=200
28 REM 'maximum' gives length of shape's
'arms'
29 REM try changing it and stepsize
30 maximum=200
40 stepsize=5
50 FOR count=0 TO maximum STEP stepsize
60 MOVE x-count,y
70 DRAW x,y+(maximum-count)
80 DRAW x+count,y
90 DRAW x,y-(maximum-count)
100 DRAW x-count,y
110 NEXT
```

'Curve-stitching' is a common activity in maths lessons in schools: many striking effects can be produced by simply connecting a series of points with straight lines. This program uses this principle by first drawing a polygon with a given number of sides and then joining each vertex to every other one:

```
10 MODE 1
20 radius=150
30 x=320:y=200
40 INPUT"How many sides has the figure got";
sides
50 CLS
60 stepsize=2*PI/sides
70 DIM x(sides),y(sides)
80 count=0
90 ORIGIN x,y
```

```
100 MOVE 0,radius
110 FOR angle=0 TO 2*PI STEP stepsize
120 DRAW radius*SIN(angle),radius*COS(angle)
130 x(count)=radius*SIN(angle):y(count)=
radius*COS(angle)
140 count=count+1
150 NEXT
155 DRAW 0,radius
160 FOR count1=1 TO sides-1
170 FOR count2=count1+1 TO sides
180 MOVE x(count1),y(count1)
190 DRAW x(count2),y(count2)
200 NEXT
210 NEXT
```

This is even more impressive if we add colour:

```
155 colour=1:DRAW 0,radius,colour
160 FOR count1=1 TO sides-1
170 FOR count2=count1+1 TO sides
173 REM experiment with the colours in line 175
174 REM change the MOD to get different effects
175 colour=1+(colour+1) MOD 3
```

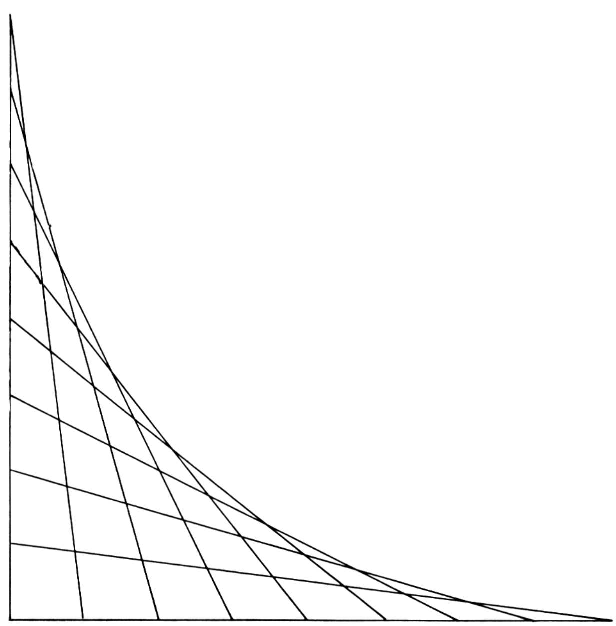


Figure 4.1 An example of curve-stitching.

```
180 MOVE x(count1),y(count1)
190 DRAW x(count2),y(count2),colour
200 NEXT
210 NEXT
```

In this chapter we shall see how much more elaborate patterns can be produced. Many of these are based round the use of the sine and cosine functions. Don't be put off by the maths — the programs are complete as they stand although there is plenty of opportunity for you to investigate their effects by substituting your own values for crucial variables.

Moire patterns

The use of the **ORIGIN** command and relative **MOVE**s makes it easy to draw symmetrical patterns on the Amstrad. This program uses the centre of the graphics area as the origin. The pattern is created by connecting points along the new x axis to points at the top and bottom of the screen. Each x coordinate is multiplied by a factor so that the lines converge or diverge:

```
10 MODE 1
19 REM use integers for speed
20 DEFINT c,f,x,y
30 xorigin=320:yorigin=200
39 REM change these factors to give a di
fferent pattern
40 factor1=3:factor2=1
50 factord=factor1-factor2
60 ORIGIN xorigin,yorigin
69 REM loop draw 4 symmetrical lines in
various positions
70 FOR count=0 TO 200
80 MOVE 0,0:MOVER count*factor1,0
90 DRAWR -count*factord,200
100 MOVER -count*factor2*2,0
110 DRAWR -count*factord,-200
120 DRAWR count*factord,-200
130 MOVER count*factor2*2,0
140 DRAWR count*factord,200
150 NEXT
```

Notice the **DEFINT** in line **20**, which helps speed the program up. It will not always be possible to define all the variables as integers, because many of the patterns in this section are produced using sine and cosine functions, which give decimal values.

The factors in line 40 can be anything you want, although avoid any greater than about 15, as this leaves the lines too far apart to produce a discernible pattern. The striking Moire patterns which appear on the screen are the result of the Amstrad leaving some pixels in the background colour, and of lines overlapping in places. Experiment with different factors, and try running the program in the other modes. You can produce a nicely woven carpet by adding these lines:

```
35 colour1=1:colour2=1
70 FOR count=0 TO 200
80 MOVE 0,0:MOVER count*factor1,0
90 DRAWR -count*factord,200,colour1
100 MOVER -count*factor2*2,0
110 DRAWR -count*factord,-200,colour2
120 DRAWR count*factord,-200,colour1
130 MOVER count*factor2*2,0
140 DRAWR count*factord,200,colour2
150 colour1=1+(colour1+1)MOD 4:colour2=1
+(colour2+1)MOD 4
160 NEXT
```

Line 150 ensures that the lines will be drawn using a cycle of the foreground colours available in mode 1. The **MOD** command gives the remainder after division: **5 MOD 4** is **1**, for example. We add one to the resulting value to avoid getting the background colour: **4 MOD 4** would otherwise produce **0**, and we would end up with a pattern containing lines that couldn't be seen against the background. In this case, lines **35** and **150** combine to give a warm orange carpet. The **PEN** colour used in the **DRAWR** commands is always either **1** (yellow) or **3** (red). The closeness of the lines (which depends on the factors in line **40**) determines whether the carpet appears orange or covered in yellow/red stripes.

By adjusting the values of the variables colour 1 and colour 2 and by changing the modulus division in line **150** to **2** or **3** a variety of effects can be achieved: patterns with diagonally opposite quarters in differing colours, or consisting of combinations of certain colours only. Running the program in other modes will give surprising results unless you adjust the range of colours that are used, as indicated in line **85**.

Lissajous figures

Lissajous figures are created using the same approach we have previously used to draw a circle. Then we kept the radius constant and took the sine and cosine of the same angle to give a point on the circumference. By varying the angle used we can draw numerous patterns:

```
10 MODE 1
20 xorigin=320:yorigin=200
30 ORIGIN xorigin,yorigin
40 MOVER 100,0
50 FOR angle=0 TO 32 STEP PI/30
60 DRAW 100*COS(angle),100*SIN(angle*0.8
)
70 NEXT
```

An alternative is to calculate the points on two curves and connect them with straight lines:

```
10 MODE 1
20 xorigin=320:yorigin=200
30 ORIGIN xorigin,yorigin
40 MOVER 100,0
50 FOR angle=0 TO 6.4 STEP PI/35
55 MOVE 200*SIN(angle),100*COS(angle)
60 DRAW 100*COS(angle),200*SIN(angle)
70 NEXT
```

Here is another example:

```
10 MODE 1
20 xorigin=320:yorigin=200
30 ORIGIN xorigin,yorigin
50 FOR angle=0 TO 6.4 STEP PI/35
55 MOVE 300*SIN(angle),50*COS(angle)
60 DRAW 10*COS(angle/5),200*SIN(angle*2)
70 NEXT
```

Let's add colour:

```
10 MODE 1
20 xorigin=320:yorigin=200
30 ORIGIN xorigin,yorigin
40 colour=1
50 FOR angle=0 TO 20 STEP PI/30
53 IF angle>10 THEN colour=3
55 MOVE 200*SIN(angle),200*COS(angle)
```

```
60 DRAW 100*COS(angle*3),200*SIN(angle/3
),colour
70 NEXT
```

As with the earlier programs, try using another mode. The sine and cosine functions which produce the patterns are largely the result of experiment and the results initially may seem rather unpredictable. However you will soon become aware of the effect of changing the variables, and you may care to investigate what happens if more complex functions are used, perhaps involving the squaring of the sine/cosine or their multiplication/division by other factors.

Spirals

We can create a spiral by taking our circle-drawing program and modifying it so that the radius constantly changes:

```
10 MODE 1
20 GOSUB 1000
100 END
1000 xorigin=315:yorigin=190
1010 ORIGIN xorigin,yorigin
1020 colour=1
1029 REM spiral's radius will increase b
y this value each time a point is plotte
d
1030 increaseradius=0.5
1040 stepsize=PI/30
1048 REM endangle determines how many ci
rcuits of spiral are drawn
1049 REM more than 40 tends to go off-sc
reen
1050 endangle=40
1059 REM spiral begins with radius of 1
1060 startradius=1
1070 GOSUB 2000
1900 RETURN
2000 MOVE 0,0
2010 FOR angle=0 TO endangle STEP stepsi
ze
2014 REM draw lines using alternate PENs
2020 DRAW startradius*SIN(angle),startra
dius*COS(angle),colour
```

```
2030 startradius=startradius+increaserad
ius
2040 NEXT
2050 RETURN
```

With the addition of a few lines we can draw several spirals inside each other:

```
1079 REM next spiral drawn within previo
us one, begins with radius of 10
1080 startradius=10
1090 GOSUB 2000
1099 REM last spiral drawn within previo
us one, begins with radius of 20
1100 startradius=20
1110 GOSUB 2000
```

The spiral can be animated so that it appears to rotate simply by plotting the points in different colours and then using the **INK** command to flash between the background and foreground colours:

```
10 MODE 1
20 GOSUB 1000
28 REM set INKs so points flash between
one colour and another
29 REM alternate flashing for points in
INKs 1 and 2
30 INK 1,1,20
40 INK 2,20,1
50 response$=""
60 WHILE response$=""
70 response$=INKEY$
80 WEND
90 INK 1,24:INK 2,20
100 END
1053 REM make both INKs used appear the
same colour
1054 REM set INK 1 to give same colour a
s INK 2
1055 INK 1,20
2014 REM draw lines using alternate PENs
2015 IF colour=1 THEN colour=2 ELSE colo
ur=1
```

This is a useful technique which we will meet again later.

Repetitive patterns

Many 'wallpaper' designs can be produced simply by replicating a figure.

```
10 MODE 1
20 GOSUB 1000
40 END
998 REM data for drawing an octagon
999 REM try some other shapes
1000 xstart=0:ystart=0
1001 REM xstart,ystart are centre of fir
st polygon drawn
1010 radius=40
1020 sides=8
1030 stepsize=2*PI/sides
1040 colour=2
1050 GOSUB 3000
1060 RETURN
2998 REM fills screen with copies of giv
en polygon
2999 REM stops when coordinate of centre
 off-screen
3000 centrex=xstart:centrey=ystart
3010 WHILE centrex<639 OR centrey<399
3020 ORIGIN centrex,centrey
3030 MOVE 0,radius
3040 FOR angle=0 TO 2*PI STEP stepsize
3050 DRAW radius*SIN(angle),radius*COS(a
ngle),colour
3060 NEXT
3069 REM shift in x direction
3070 centrex=centrex+radius*2
3079 REM if it's off-screen, move up and
 start again
3080 IF centrex>639 AND centrey<399 THEN
 centrex=xstart:centrey=centrey+radius*2
3090 WEND
3100 RETURN
```

More elaborate results occur if a second set of figures are superimposed on the first — these may be completely different figures or larger or smaller versions of the first:

```
30 GOSUB 2000
1999 REM data for drawing a hexagon
2000 xstart=40:ystart=40
2010 radius=40
2020 sides=6
2030 stepsize=2*PI/sides
2040 colour=2
2050 GOSUB 3000
2060 RETURN
```

The rows can also be staggered by beginning each row in one of two alternative positions:

```
3079 REM if it's off-screen, move up and
 start again
3080 IF centrex>639 AND centrey<399 THEN
 GOSUB 4000
3090 WEND
3100 RETURN
3999 REM stagger start of each row of po
lygons
4000 IF xstart=0 THEN xstart=radius ELSE
 xstart=0
4010 centrex=xstart:centrey=centrey+radi
us*2
4020 RETURN
```

Rotating shapes

It is a relatively simple task to modify the circle-drawing program so that it can draw any polygon. This can be incorporated as a subroutine into a program that will enlarge and rotate a given basic shape, creating a spiral pattern:

```
10 MODE 1
20 GOSUB 1000
30 GOSUB 1500
40 END
999 REM read data for polygon
1000 READ sides
1010 READ radius
1020 READ centrex,centrey
1030 READ radiuschange,anglechange
1040 colour=2
```

```
1050 stepsize=2*PI/sides
1060 startangle=0:finishangle=2*PI
1070 RETURN
1500 ORIGIN centrex,centrey
1508 REM keep drawing polygon until it's
 too big
1509 REM for all sample polygons in data
 this is when radius>200
1510 WHILE radius<200
1520 MOVE radius*SIN(startangle),radius*
COS(startangle)
1530 FOR angle=startangle TO finishangle
 STEP stepsize
1540 DRAW radius*SIN(angle),radius*COS(a
ngle),colour
1550 NEXT
1560 DRAW radius*SIN(startangle),radius*
COS(startangle)
1569 REM increase radius
1570 radius=radius+radiuschange
1579 REM rotate start so next polygon is
 at an angle to this one
1580 startangle=startangle+anglechange
1590 finishangle=finishangle+anglechange
1600 WEND
1610 RETURN
2000 DATA 3,20,300,200,5,1
2010 DATA 3,20,300,200,3,10
2020 DATA 4,30,300,200,4,3
2030 DATA 6,20,300,200,1,6
2040 DATA 6,20,300,200,6,1
2050 DATA 8,10,300,200,5,10
2060 DATA 8,10,300,200,5,2
```

Run the program as it stands. The **DATA** lines from **2000** onwards give data for a variety of figures. Delete the first **DATA** line and run the program again to see the effect of varying the number of sides in the figure and the rate at which it is enlarged and rotated. Repeatedly delete the new first **DATA** line and re-run the program to see the remainder of the examples.

Some very impressive effects can be achieved by judicious use of colour:

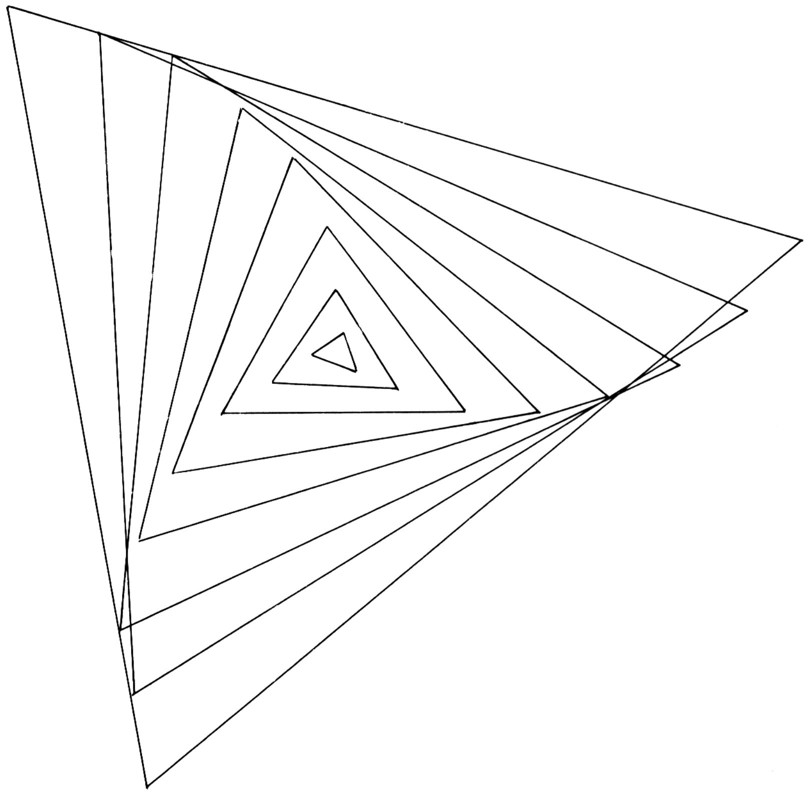


Figure 4.2 The type of pattern that can be produced by rotating a basic shape.

```
10 MODE 1
20 GOSUB 1000
29 REM alternate flashing colours for po
ints in INKS 2 AND 3
30 GOSUB 1500
40 INK 2,1,20
50 INK 3,20,1
59 REM wait for key depression
60 response$=""
70 WHILE response$=""
80 response$=INKEY$
90 WEND
99 REM restore to normal
100 INK 2,20
110 INK 3,6
120 END
1504 REM set INK 3 same as INK 2
```

```
1505 INK 3,20
1594 REM alternately draw each polygon u
sing INK 2 or INK 3
1595 IF colour=2 THEN colour=3 ELSE colo
ur=2
```

Pressing any key once the figure has been drawn gives a really mind-boggling effect as the colours switch between foreground and background!

Exercises

1) Create a modified version of the polygon vertex-joining program by drawing two polygons and joining their vertices.
2) Investigate the effect of the following on the Moire pattern program: changing the STEP of the FOR. . .NEXT loop; superimposing a second pattern using different factors and colours; shifting the origin to create a patchwork of patterns next to each other.
3) Modify the rotating polygon program so that the number of sides of the polygon varies during the course of the program. Alternate between, for example, a triangle and a pentagon; or cycle through the polygons, adding a new side with each rotation of the figure until a limit of say 20 sides is reached when the cycle begins again.

Sketching on the screen

An alternative to letting the computer create the pattern is to devise a program that allows the user to draw and manipulate shapes on the screen, and this is the topic we shall examine in the remainder of the chapter. To make the software relevant to the widest possible audience, all line-drawing etc. is carried out from the keyboard, but the programs that follow are readily modified if you wish to use joysticks.

Throughout this section we shall use mode 0 as it offers the greatest range of colours. Our first module contains the most essential feature of any line-drawing program: it enables us to plot points on the screen and draw lines in different directions:

```
10 MODE 0
20 x=320:y=200
30 colour=1
40 MOVE x,y
50 PLOT x,y,colour
60 GOSUB 1000
70 END
997 REM scan keyboard - end if 'e' press
ed
1000 WHILE response$<>"e"
1010 response$=INKEY$
1019 REM draw line commands: up/down=a/z
, left/right=,/.
1020 IF response$="a" THEN y=y+2
1030 IF response$="z" THEN y=y-2
1040 IF response$="," THEN x=x-4
1050 IF response$="." THEN x=x+4
1060 PLOT x,y,colour
1070 WEND
1080 RETURN
```

A point is initially plotted at the coordinates specified in line 40. The INKEY$ in line 60 checks the keyboard to see if any key is currently being pressed. (Movement is indicated by depression of **a/z** for up/down, and **,/,** for left/right.) The coordinates of the point are updated accordingly, and the position of the new point is plotted.

Note that the structure of this brief program makes it very easy to add extra commands. Movement of the point is presently confined to up/down/left/right, but by adding lines within the loop from 50 to 110 we could make diagonal movement possible.

As it stands, the program only allows continuous lines, and it is impossible to move to a new position without drawing a line. We can add options so that the point can be plotted in either the foreground or background colour by pressing '**f**' or '**b**' respectively:

```
1051 IF response$="f" THEN colour=1
1052 IF response$="b" THEN colour=0
```

Unfortunately once we select the background colour, the point is no longer visible, which makes it difficult to move to a new position with any confidence! We can get round this by plotting the point twice:

```
35 foregroundcolour=1
1005 PLOT x,y,colour
1060 PLOT x,y,foregroundcolour
```

If the background colour is selected, the point at the present x,y position will be plotted invisibly at line **55**, and then again in the foreground colour at line **100**. As a result, the point flashes on and off, and acts as a cursor identifying its present position.

We can allow a change of foreground colour either by selecting a colour by the depression of a particular key, or by cycling through the colours as we did in an earlier program:

```
1053 IF response$="c" THEN colour=1+(col
our+1) MOD 3
```

This gives a choice of just three of the foreground colours, all of which produce easily visible lines.

It is sometimes difficult to see the colour of the point, and the accurate plotting of rectangles and the like is not straight-forward using only the naked eye to gauge distances. We could simply print this information on the screen, but we shall take this opportunity to introduce the **WINDOW** command, which confines all text to a specified area:

```
15 WINDOW 1,40,24,25
16 PRINT "Colour: "
17 PRINT "x:        y: ";
```

The four numbers following the **WINDOW** command specify first the x text coordinates making up the left and right boundaries of the window, and then the y text coordinates comprising the top and bottom of the window. In this case all further text will be printed to two lines at the bottom of the screen:

```
1051 IF response$="f" THEN colour=foregr
oundcolour
1052 IF response$="b" THEN foregroundcol
our=colour:colour=0
1053 IF response$="c" THEN colour=1+(col
our+1) MOD 3
```

```
1060 PLOT x,y,foregroundcolour
1065 GOSUB 2000
1070 WEND
1080 RETURN
2000 IF oldcolour<>colour THEN LOCATE 9,
1:PRINT colour
2010 IF oldx<>x THEN LOCATE 4,2:PRINT x;
2020 IF oldy<>y THEN LOCATE 12,2:PRINT y
;
2030 oldcolour=colour
2040 oldx=x:oldy=y
2050 RETURN
```

Lines can now be accurately positioned on-screen, because the program supplies a continual update of the present **PEN** colour of the point and its x and y coordinates. Note that the Amstrad still considers the text window to be part of the graphics screen. You will find that you can plot points over the text!

There are still a number of flaws in the program. Lines can only be erased by drawing them again in the background colour. This gives rise to another problem when the point, set to the background colour, crosses a line that has already been drawn — part of the line is erased. This might seem an intractable problem, but is surprisingly easy to solve, although we shall have to call on our knowledge of binary to do it.

Exercises

1) The drawing program does not contain any checks to see if the point plotted is off the screen. Modify the program so that it is impossible to move off the screen or draw in the text window.
2) Extend the range of colours that can be used to 16. Use the **INK** command so that the colours that can be displayed are not just the default colours for mode 0, but are initially chosen from the full 27 colours available.
3) Add some extra commands so that diagonal lines can be drawn.

Using EXOR

In an earlier chapter I commented on the value of an understanding of binary/hex, and we shall see now and later in the book why this is especially relevant where many graphics operations are concerned.

It is important to remember that the screen display itself is actually a representation of part of the computer memory. All the characters printed, and each graphics line drawn is produced as a result of particular binary values being stored in the memory locations which the Amstrad examines to construct the screen display.

Any line drawn on the screen causes the bytes (the 8-bit binary values) at the relevant memory locations to be changed. The value at a location determines whether a pixel should be lit or unlit, and if lit, what colour it should be. In fact, the colours for each pixel are derived from an individual byte in a way which is not obvious, but which need not concern us here. For the purposes of the discussion which follows we shall use a simplified model of the byte/pixel relationship, and assume that the value of a single byte stored in memory indicates to the Amstrad the value of a single pixel to be displayed on the screen.

Suppose that we are dealing with only four colours, and the bytes representing the screen memory thus all have one of the four values shown in Figure 4.3.

Binary code	Colour
00000000	Blue
00000001	White
00000010	Yellow
00000011	Red

Figure 4.3 Possible binary representations the computer might use to indicate four different colours.

If the screen was completely blue, all bytes would be **00000000**; if it was completely white, they would all be **00000001**, and so on. Drawing a white line on the screen has the effect of changing the bytes at the positions concerned so that they are all **00000001** drawing a yellow line at the same position makes their value **00000010**.

Earlier we noted that the lower ASCII codes give special commands to the Amstrad, such as 'Move the cursor back one space' or 'Turn off the screen'. One of these codes influences

the way in which the Amstrad treats graphics points.
The Amstrad can be set so that it plots any points on-screen using the Exclusive Or option (EXOR or XOR for short). Without this option, the Amstrad simply replaces the old value bytes concerned by the new value. For a line drawn in yellow, all the bytes on the line become 00000010, for example. Using the EXOR option makes the Amstrad plot all points by combining the old value of each byte with the new value according to a fixed set of rules.

Let's examine an individual byte on the screen so that we can see why the program produces the results that it does. To begin with, the byte has the value 00000000, i.e., it indicates a pixel in the background colour, blue, as in Figure 4.4. A line is drawn across this point in yellow (a byte of 00000010), as in Figure 4.5.

```
00000000          A point in the background colour, blue.
```

Figure 4.4

```
00000000          A blue point and
00000010          a yellow line crossing that point
```

Figure 4.5

Because the Amstrad has the EXOR option set, it does not simply replace the byte 00000000 (for blue) by the byte 00000010 (for yellow). Instead it combines the value of the two bytes. If corresponding bits are different, the result is 1; if corresponding bits are the same, the result is zero, so we end up with a yellow point, precisely what we would have expected had we not used the EXOR option!

```
    00000000      A blue point and
EXOR00000010      a yellow line crossing that point
    00000010      gives a yellow point
```

Figure 4.6

However, suppose we now plot an identical yellow line over the line we have just drawn. The effect this time is rather unexpected, see Figure 4.7. Because the bits making up the old byte and the new byte are exactly the same, the EXOR option results in a byte of 00000000 — i.e., the line disappears, as it has been drawn in blue, the background colour. Drawing

	00000010	A yellow point and
EXOR	00000010	a yellow line crossing that point
	00000000	gives a point in the background colour, blue.

Figure 4.7

another yellow line returns us to the situation of Figure 4.4, and the line will reappear.

	00000001	A white point
EXOR	00000010	crossed by a yellow line

Figure 4.8

What happens if a yellow line is drawn on top of a white line? The EXOR option results in a red line, as in Figure 4.9. But again, drawing the line a second time in the same colour restores everything to its original state, as in Figure 4.10.

	00000001	A white point
EXOR	00000010	crossed by a yellow line
	00000011	gives a red point

Figure 4.9

	00000011	A red point
EXOR	00000011	crossed by a red line
	00000000	gives a point in the background colour, blue.

Figure 4.10

The EXOR option might not seem very useful, but the graphics effects we have just seen are invaluable in any drawing program. Lines can be drawn and erased without affecting other lines; we can experiment and move lines to various positions before fixing them permanently using normal drawing. All that we need do is ensure that any temporary line or point is plotted twice using EXOR so that all its traces are removed.

Drawing diagrams

Before we begin to design a new drawing program to take advantage of the graphics **EXOR** option, let's summarise the extra features that could usefully be included:

1) lines can be temporary or fixed permanently;
2) lines can be erased;
3) drawings can be saved;
4) standard shapes such as circles, rectangles, etc. can be drawn;
5) parts of the drawing can be translated, enlarged or otherwise transformed.

The last two features will be discussed in a later chapter. We have just discovered how to draw temporary lines, so 1) presents no problem. The easiest way to implement both 2) and 3) is by saving the coordinates of all fixed lines in an array. This makes it simple to identify a line and delete it. It also makes it easy to save the drawing to a file: we simply save the list of coordinates stored in the array and use them to reproduce the drawing at a later date.

We will use a modular approach so that the program can be extended without problem:

```
10 MODE 0
20 DIM x(100),y(100)
30 startx=320:starty=200
40 x=320:y=200
50 foregroundcolour=1
60 PRINT CHR$(23);CHR$(1);
70 GOSUB 1000
80 GOSUB 2000
90 END
999 REM draw line
1000 PLOT x,y,foregroundcolour
1010 DRAW startx,starty
1020 RETURN
2000 WHILE response$<>"e"
2010 GOSUB 1000
2020 response$=LOWER$(INKEY$)
2030 IF response$="a" THEN y=y+2
2040 IF response$="z" THEN y=y-2
2050 IF response$="," THEN x=x-4
```

```
2060 IF response$="." THEN x=x+4
2070 IF response$=" " THEN GOSUB 3000
2080 GOSUB 1000
2090 WEND
2100 RETURN
2999 REM set graphics to normal to draw
permanently
3000 PRINT CHR$(23);CHR$(0);
3010 GOSUB 1000
3019 REM back to EXOR drawing
3020 PRINT CHR$(23);CHR$(1);
3030 count=count+1
3040 x(count)=x:y(count)=y
3050 startx=x:starty=y
3060 RETURN
```

Line **20** sets up two arrays to hold the coordinates for up to 100 points. (This figure is arbitrary and may be increased.) The coordinates of the present point are given by the values of the variables x and y. The values of start x and start y give the coordinates of the last point 'fixed'. These coordinates must be available so that we can draw a permanent line between the two points if we wish.

Subroutine **2000** is the driving module of the program. It scans the keyboard for input and successively draws and erases a line from (x, y) to (startx, starty).

If you run the program you will find that, as you move the point around, the Amstrad draws a flickering yellow line from it to the point (startx, starty), subroutine **1000**. This technique of allowing a line to be stretched is referred to as 'rubber-banding' — you can probably see why!

Pressing the Space Bar fixes the line permanently via subroutine **3000**. Line **3000** returns the graphics drawing mode to normal, draws the line permanently, and then switches back to the EXOR option for drawing to continue. The coordinates x and y of the present point are stored in the arrays x() and y(), and startx and starty are given new values so the next line will be drawn from where the current one ends.

We can modify the program so that we can chose whether to draw a line or not:

```
55 linedraw=0
```

```
999 REM draw line (or not, if linedraw=0
)
1000 PLOT x,y,foregroundcolour
1005 IF linedraw=0 THEN RETURN
1010 DRAW startx,starty
1020 RETURN
2070 IF response$=" " THEN GOSUB 3000:li
nedraw=1
2071 IF response$="l" THEN IF linedraw=0
 THEN linedraw=1 ELSE linedraw=0
```

Line **2071** uses the '**l**' key as a toggle to switch between line-drawing and no line-drawing. When **linedraw=0** no line is drawn, as subroutine **1000** terminates before (x,y) can be joined to (startx, starty). Running the program now reveals that we no longer have to draw the first line beginning at (startx,starty), but can move the point to anywhere we wish before 'fixing' it by pressing '**f**'.

Deleting any line is a little tricky. We shall instead only allow the deletion of the last line drawn, indicated by pressing the '**d**' key:

```
2072 IF response$="d" THEN GOSUB 4000
3998 REM doesn't work if you to try to d
elete a non-existent line
3999 REM we'll sort that out in the next
 program!
4000 x=x(count):y=y(count)
4010 count=count-1
4020 startx=x(count):starty=y(count)
4030 GOSUB 1000
4040 RETURN
```

This does allow deletion of any line, but only at the cost of deleting all lines drawn subsequently.

If you find the keyboard response a bit sluggish, you might like to add the following lines:

```
1 DEFINT c,f,l,o,s,x,y
2 SPEED KEY 2,2
89 SPEED KEY 10,5
```

We have noted before that using integers speeds a program up. **SPEED KEY** is followed by two numbers, the start delay and the auto-repeat period, in 1/50th second units. These two values determine how rapidly the Amstrad responds to a key depression. When a key is pressed, the computer waits for a time equal to the start delay before repeating the character. Thereafter it is repeated at intervals governed by the auto-repeat period. It is vital that you set **SPEED KEY** to normal at the end of the program. Over-rapid response to key presses can make it virtually impossible to type a coherent instruction!

Having created a drawing masterpiece it would be nice to save it. To do so we need to modify the program slightly. Recording the coordinates of all the points is no longer adequate, as we also need to know whether a point is joined to the previous one or not:

```
20 DIM x(100),y(100),l(100)
3030 count=count+1
3040 x(count)=x:y(count)=y
3045 l(count)=linedraw
3050 startx=x:starty=y
3060 RETURN
4000 x=x(count):y=y(count):linedraw=l(co
unt)
```

The arrays x() and y() hold the coordinates of all the points; a third array l() is introduced to indicate whether a point is joined to the previous one. Whenever a point is fixed, l(count) is used to record the current condition of `linedraw`. If this is zero, line-drawing has been switched off, and the current point is not connected to the one before. Any value other than zero shows that line-drawing is switched on and the present point must be connected to the earlier one.

```
2074 IF response$="i" THEN GOSUB 7000
2075 IF response$="o" THEN GOSUB 8000
6000 CLG
6010 startx=x(1):starty=y(1)
6020 FOR value=2 TO count
6030 x=x(value):y=y(value)
6040 linedraw=l(value)
6050 GOSUB 1000
6060 startx=x:starty=y
6070 NEXT
6080 x=320:y=200:linedraw=0
6090 RETURN
7000 MODE 1
```

```
7010 PRINT"To load data from a coordinat
e file"
7020 INPUT"What is the file name";file$
7030 OPENIN file$
7040 count=0
7050 WHILE NOT EOF
7060 count=count+1
7070 INPUT #9,x(count),y(count),l(count)
7080 WEND
7090 CLOSEIN
7100 MODE 0
7110 WINDOW 1,20,25,25
7120 startx=x(count):starty=y(count)
7130 x=startx:y=starty
7140 GOSUB 6000
7150 RETURN
8000 MODE 1
8010 PRINT"To save a picture to a file"
8020 INPUT"Please name the file.",file$
8030 OPENOUT file$
8040 counter=0
8050 WHILE counter<=count
8060 WRITE #9,x(counter),y(counter),l(co
unter)
8070 counter=counter+1
8080 WEND
8090 CLOSEOUT
8100 MODE 0
8110 WINDOW 1,20,25,25
8120 GOSUB 6000
8130 RETURN
```

Subroutine **8000** writes all the coordinate and connection data from the three arrays to a file, and then uses subroutine **6000** to recreate the drawing. If we can save the data, we also need to be able to load it back in, and this is handled by subroutine **7000**. There is no need to switch to mode 1 in these subroutines, but doing so makes all the messages easier to read! The load routine is called on depression of '**i**' (input from a file) and the save routine is called on depression of '**o**' (output to a file). You may prefer to substitute your own keys.

Our basic drawing program is almost complete. Let's just add a colour option:

```
2073 IF response$="s" THEN GOSUB 5000
5000 WINDOW 1,20,25,25
5010 INPUT"Scale";scale
5018 REM scale all values by scale facto
r
5019 REM make present cursor coordinates
 new screen centre
5020 FOR value=1 TO count
5030 x(value)=scale*(x(value)-x)+320
5040 y(value)=scale*(y(value)-y)+200
5050 NEXT
5060 GOSUB 6000
5070 RETURN
```

Adding colour does create a problem, because to delete the same line it must be drawn again using **EXOR** and the correct colour. Otherwise, as we saw earlier, a white line drawn on a yellow line will not erase the original line but merely change its colour! Fortunately we don't need to set up yet another array for the colour — this information is already stored in l() and can be used when lines are deleted or when a picture is drawn using data loaded from a file.

Let's conclude the chapter with a demonstration of the flexibility of the core program. The addition of the following routine enables you to create a drawing, then enlarge it (or reduce it) so that greater detail can be added:

```
2070 IF response$=" " THEN GOSUB 3000:li
nedraw=foregroundcolour
2071 IF response$="l" THEN IF linedraw=0
 THEN linedraw=foregroundcolour ELSE lin
edraw=0
2076 IF response$="c" THEN foregroundcol
our=1+(foregroundcolour+1) MOD 3
3045 IF linedraw>0 THEN l(count)=foregro
undcolour
4025 IF linedraw>0 THEN foregroundcolour
=linedraw
```

Pressing '**s**' calls subroutine **5000**, which requests a scale factor by which the drawing is to be enlarged or reduced. For example, typing '**2**' will cause the picture to be redrawn at twice its present size, '**0.1**' reduces it to a tenth of its size, and so on.

The routine takes advantage of the fact that the Amstrad will accept x and y coordinates for **PLOT**, **MOVE** and **DRAW** commands even when these coordinates lie beyond the screen boundary. The computer will 'draw' these lines, but they will only be seen if they happen to cross the graphics area depicted on the screen — i.e., some of the points on the line have x coordinates lying between 0 and 639 and y coordinates between 0 and 399.

Once you have chosen a scale factor, the Amstrad multiplies all the coordinates in the arrays x() and y() by that scale factor. The current position of the cursor is used as the centre of enlargement. The new drawing is centred around **(320,200)**, which is also taken as the new position of the cursor.

Using integers may appear to be a good way of speeding up the program, but it is a disadvantage here as it limits the choice of scale factors to whole numbers only.

Exercises

1) Add a few lines at the end of the drawing program to restore key response speed etc. to normal.
2) Introduce checks so that the drawing program does not allow movement off-screen.
3) The 'delete line' routine has a flaw in that it is possible to delete lines back to the starting position, at which point the program crashes. Introduce checks to prevent this.
4) Add a routine to the program to allow the user to select a comfortable start delay and auto-repeat for the keys.
5) Add a routine to the program so that the coordinates of the cursor are continuously displayed on the screen.
6) (*more difficult*) Introduce selective deletion of lines so that lines other than the last can be erased. (You will need to cycle through all the lines erasing/redrawing them at the touch of a key until the required line is reached, which should then be permanently deleted. Remember to reflect this line deletion in the data stored in the arrays, otherwise the line will magically reappear when the array values are loaded back from a file!)

Chapter Five

Animation . . .

Moving line-drawings

In the last chapter EXOR was introduced, and we saw how it could be utilised in the drawing/deleting of lines. We shall now see how EXOR and related commands can be used to improve the quality of animation.

One method of animation is to successively draw and then delete a figure from the screen. This is the most primitive approach, but provided the figure concerned is not too complex, the speed of the computer produces an acceptable result. This program moves a rectangle across the screen by drawing and then deleting it at each position:

```
10 MODE 1
20 x=100:y=100
30 xdistance=50:ydistance=100
39 REM move units in the x direction eac
h time the rectangle is drawn
40 xinc=4
50 WHILE x<639
59 REM draw the rectangle
60 colour=1:GOSUB 1000
69 REM delete the rectangle
70 colour=0:GOSUB 1000
80 x=x+xinc
90 WEND
100 END
999 REM commands to draw rectangle
1000 MOVE x,y
1010 DRAWR xdistance,0,colour
1020 DRAWR 0,ydistance
1030 DRAWR -xdistance,0
1040 DRAWR 0,-ydistance
1050 RETURN
```

A simple modification moves the rectangle diagonally:

```
40 xinc=4:yinc=2
80 x=x+xinc:y=y+yinc
```

With the addition of another couple of lines we can even 'bounce' the figure around the screen:

```
45 continue=1
50 WHILE continue=1
59 REM draw the rectangle
60 colour=1:GOSUB 1000
69 REM delete the rectangle
70 colour=0:GOSUB 1000
79 REM update x,y coordinates and check
if they are off-screen
80 x=x+xinc:y=y+yinc
81 IF x<0 OR x>639 THEN xinc=-xinc:x=x+2
*xinc
82 IF y<0 OR y>399 THEN yinc=-yinc:y=y+2
*yinc
90 WEND
```

The results are not so good when more lines are involved. Suppose we try to animate a line-drawing of a dog. To bring some life to the program we draw the figure in two slightly different positions. The computer will first draw the dog in one position, then delete it, then draw the dog in the alternative position. After deleting the second image the coordinates are updated and the whole cycle is repeated. To simplify the alternation between the two sets of image coordinates it is easiest to store them in an array:

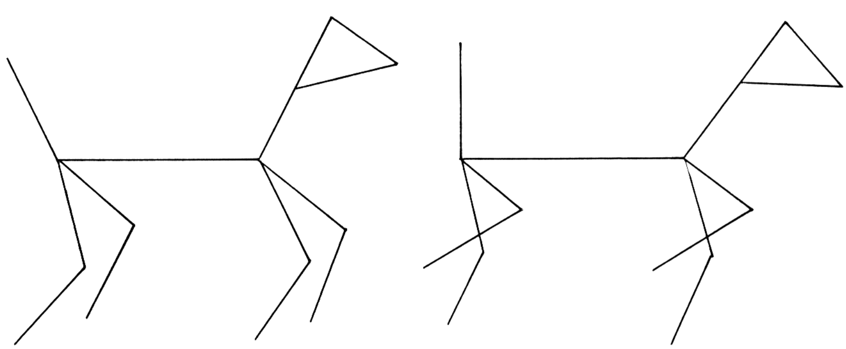


Figure 5.1 Using the same figure in two different positions to improve animation.

```
10 MODE 1
20 GOSUB 1000
30 GOSUB 2000
40 END
998 REM read in 2 sets of 26 coordinates
999 REM for drawing 2 images of the dog
1000 DIM x(100),y(100)
1010 FOR count=1 TO 52
1020 READ x(count),y(count)
1030 NEXT
1040 RETURN
1050 DATA 0,0,20,40,20,40,10,80,10,80,0,
120,20,10,35,50,35,50,10,80,10,80,70,80
1060 DATA 60,0,80,40,80,40,70,80,80,10,9
5,50,95,50,70,90,70,80,90,140,90,140,110
,120,110,120,80,110
1070 DATA 5,10,25,40,25,40,15,80,15,80,1
0,120,0,15,30,50,30,50,15,80,15,80,75,80
1080 DATA 75,80,85,40,85,40,65,10,75,80,
90,50,90,50,60,15,75,80,100,130,100,130,
120,110,120,110,90,100
1999 REM successively draw/delete dog
2000 xinc=0
2010 flag=0
2020 WHILE x(1)+xinc<639
2030 IF flag=0 THEN start=1:flag=1 ELSE
start=27:flag=0
2039 REM draw dog
2040 colour=1:GOSUB 3000
2049 REM delete dog
2050 colour=0:GOSUB 3000
2059 REM move to new position to draw ne
xt dog
2060 xinc=xinc+20
2070 WEND
2080 RETURN
2999 REM draw dog as a series of connect
ing points
3000 FOR count=start TO start+25 STEP 2
3010 MOVE x(count)+xinc,y(count)
3020 DRAW x(count+1)+xinc,y(count+1),col
our
3030 NEXT
3040 RETURN
```

In this case the whole process is too slow and it is clear that the figure is being deleted and then redrawn to another position. A different approach is required.

We can make use of a facility we touched on in Chapter 3: the ability of the computer to change the range of colours available in a mode by using the **INK** command. If we draw a figure using a **PEN** that has been set to the background colour, the figure can be made to appear instantly simply by switching the **INK** to the foreground colour.

The following program uses this method to display alternately two rectangles drawn at different positions on-screen. Every time a key is pressed, the **INK** colours are changed so that the previously displayed rectangle is set to the background colour and the other rectangle to the foreground colour so that it becomes visible:

```
10 MODE 1
20 x=100:y=100
30 xd=50:yd=100
39 REM set PEN 1 to background colour
40 INK 1,1
49 REM draw one rectangle using PEN 1 in
 background colour
50 colour=1:GOSUB 1000
60 x=300:y=300
70 xd=100:yd=50
79 REM set PEN 2 to background colour
80 INK 2,1
89 REM draw second rectangle using PEN 2
 in background colour
90 colour=2:GOSUB 1000
100 continue=1
109 REM repeatedly display until 'e' pre
ssed
110 WHILE response$<>"e"
119 REM switch PEN 1 to foreground colou
r, PEN 2 to background
120 INK 1,24
130 INK 2,1
139 REM wait for key depression
140 response$=""
150 WHILE response$=""
160 response$=LOWER$(INKEY$)
```

```
170 WEND
179 REM switch PEN 2 to foreground colou
r, PEN 1 to background
180 INK 1,1
190 INK 2,24
200 response$=""
209 REM wait for key depression
210 WHILE response$=""
220 response$=LOWER$(INKEY$)
230 WEND
240 WEND
249 REM restore normal colours
250 INK 1,24
260 INK 2,20
270 END
1000 MOVE x,y
1010 DRAWR xd,0,colour
1020 DRAWR 0,yd
1030 DRAWR -xd,0
1040 DRAWR 0,-yd
1050 RETURN
```

This gives a clear display because the figures have already been drawn and are displayed instantly. We could extend this idea and animate a sequence of drawings by drawing them in the background colour and then successively displaying them. In this a method we could use to animate the line-drawing of the dog? Not as it stands, because when the figures overlap, we have a problem, as you can see by running this program:

```
59 REM this time second rectangle overla
ps
60 x=120:y=100
```

Part of one rectangle is missing where the figures overlap. This is because the line is present in both figures, and setting the **INK** for one rectangle to the background colour causes it to disappear for the other.

Providing there is no overlap of lines, changing the **INK**s gives smooth and rapid animation. It is especially useful in mode 0, where we have 16 different colours. Up to 15 different images can be drawn in the background colour and then displayed in succession by switching the relevant **INK** to the foreground colour and then to the background colour once again. For example, here we have a figure which expands and then shrinks again:

```
10 MODE 0
19 REM start position for corner of rect
angle
20 startx=260:starty=180
29 REM length of its sides
30 lengthx=20:lengthy=20
39 REM difference in size for successive
 rectangles
40 incx=8:incy=6
50 startink=1:endink=15
60 GOSUB 1000
70 GOSUB 2000
80 END
998 REM draw 15 rectangles in background
 colour
999 REM put one inside the other
1000 FOR count=startink TO endink
1010 INK count,1
1020 movex=count*incx
1030 movey=count*incy
1040 sidex=lengthx+2*movex
1050 sidey=lengthy+2*movey
1060 MOVE startx-movex,starty-movey
1070 DRAWR sidex,0,count
1080 DRAWR 0,sidey
1090 DRAWR -sidex,0
1100 DRAWR 0,-sidey
1110 NEXT
1120 RETURN
1999 REM cycle through INKs displaying o
ne rectangle at a time
2000 continue=1
2010 startink=1:nextink=2
2019 REM continue forever
2020 WHILE continue=1
2029 REM wait for any key depression
2030 response$=""
2040 WHILE response$=""
2050 response$=INKEY$
2060 WEND
2069 REM switch previous rectangle to ba
ckground
```

```
2070 INK startink,1
2079 REM switch next rectangle to foregr
ound
2080 INK nextink,24
2088 REM increment INKs so on next cycle
 present INK
2089 REM becomes background and new INK
becomes foreground
2090 startink=(startink+1) MOD 16
2100 IF startink=0 THEN startink=1
2110 nextink=(nextink+1) MOD 16
2120 IF nextink=0 THEN nextink=1
2130 WEND
2140 RETURN
```

Using this method is particularly effective if the display is cyclical, as in this case. It does have its limitations, however — avoiding overlapping lines is not always possible.

We have already seen an example of an instance where drawing a line does not disturb any lines already present — we use the **EXOR** option. Perhaps the same approach will work here:

```
15 PRINT CHR$(23)CHR$(1);
265 PRINT CHR$(23)CHR$(0);
```

This is a partial success — both rectangles are completely visible, but the shared points appear in the wrong colour, red. We can see the reason for this result if we examine the way in which colours are represented in mode 1.

Only four colours can be displayed simultaneously in mode 1, because the colour of each point is determined by a 2-bit code. As there are just four differing combinations of two bits, only four colours are possible. These codes will never change, although we could use the **INK** command so that the computer would interpret any of the codes as indicating a different colour. For example, it would display a pixel coloured red rather than yellow if we used an **INK 1,6** command.

00000000	Blue	Effectively, the colour at any point is shown using a 2-bit code, and this is used from now on.
00000001	Yellow	
00000010	Bright cyan	
00000011	Red	

Figure 5.2 The binary codes for the four colours in mode 1.

In this particular case we are drawing two rectangles, one using INK 1 (code Ø1) and the other INK 2 (code 1Ø). Remember the effect of EXOR. It combines the old bit combination with the new one according to a simple rule: if the bits are the same the result is Ø; if the bits are different the result is 1. Where the two rectangles overlap we will get the result shown in Figure 5.3. Code 11 indicates PEN 3, normally set to red in mode 1, and so the line is displayed in red. This

	Ø1	A point on a yellow line
EXOR	1Ø	crossed on a cyan line
	11	gives a red point

Figure 5.3

suggests a way around our problem. If we use the INK command to re-set PEN 3 so that it produces yellow rather than red, the overlapping area will no longer be visible:

```
11 REM set INK 3 to yellow
12 INK 3,24
261 REM restore INK 3 to normal at end
262 INK 3,6
```

Although the rectangles overlap, they can both be displayed alternately by switching the INK colours. This forms the basis for an approach which produces much smoother animation:

1) The first figure of the animation is drawn in the foreground colour and displayed;
2) The second figure is drawn using a PEN that has been set to the background colour;
3) The INKs are switched so that the second figure is displayed and the first hidden;
4) The first figure is deleted from its old position;
5) A new figure is drawn in the background colour;
6) The INKs are switched so that the new figure is displayed and the second hidden;

. . . and so on, until the animation is complete. The only difficulty arises when the figures overlap in places: we then need to draw or erase one figure without affecting the other. Let's examine this problem more closely.

We will suppose that we are working in mode 1, and the two successive figures are drawn using PEN 1 and PEN 2 respectively. This gives us this interpretation of the meaning of the four 2-bit colour codes. We do not need to concern ourselves with the actual colours that will be shown on the screen, as the INK command enables us to choose the colour which will be produced by any particular PEN. We will instead concentrate on these 2-bit codes, and the effects we wish to have on them.

To erase any line drawn using PEN 1, we need to convert the code 01 to 00, the background colour code. We could do this by EXORing every point on the line with 01. However, some points on the line might overlap points with a line from the second figure, and so have the code 11. EXORing these points with 01 gives the result shown in Figure 5.5. Any overlap point would end up in the colour for PEN 2, which is what we want. EXOR only seems the answer however — it fails in one situation. If a point lies at the intersection of two lines, erasing one line with EXOR will delete the point, but erasing the second line with EXOR will bring it back again!

	01	A point on a yellow line
EXOR	01	crossed by a yellow line
	00	gives a point in the background colour, blue

Figure 5.4

	11	A point on both lines
EXOR	01	crossed by a yellow line
	10	gives a point on the cyan line

Figure 5.5

We could achieve the correct result a slightly different way, so we take this opportunity to introduce yet another of the line-drawing modes available on the Amstrad.

The computer can be set so that it ANDs the code for a point with its new code. This is achieved by using the control code PRINT CHR$(23) CHR$(2); which causes all subsequent graphics commands to use AND.

	01001101
AND	11100100
	01000100

Figure 5.6 An example of the effect of AND.

The effect of AND is that a bit is set to 1 only if the corresponding bits in the first AND second code are both 1;

otherwise the code is set to 0. We can delete points drawn with PEN 1 by ANDing with the code 10, as in Figure 5.7. This also leaves overlap points in the correct colour, as in Figure 5.8. Similarly, points drawn with PEN 2 can be deleted by ANDing with the code 01, as in Figure 5.9.

	01	A point on a yellow line
AND	10	crossed by a cyan line
	00	gives a point in the background colour, blue.

Figure 5.7

	11	A point on both lines
AND	10	crossed by a cyan line
	10	gives a point on the cyan line.

Figure 5.8

	10	A point on a cyan line
AND	01	crossed by a yellow line
	00	gives a point in the background colour, blue.

Figure 5.9

Let us now turn to drawing without disturbing points already plotted. Suppose we are drawing using PEN 1. The desired results at various points are shown in Figure 5.10.

Colour code at point	Desired code after line drawn with PEN 1 passes through point
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0	0 0 0 0 0 0 1 1
0 0 0 0 0 0 1 1	0 0 0 0 0 0 1 1

Figure 5.10

Neither EXOR nor AND gives the required outcome. There is a further graphics drawing mode which we have not yet met — the OR mode. This is set by the control codes PRINT CHR$(23) CHR$(3); and gives a bit result of 1 if either one bit OR its corresponding bit is 1. We can draw with PEN 1 by ORing with the code 01. We can draw with PEN 2 by ORing with the code 10, as in Figure 5.12.

We can now identify a sequence of graphics-drawing modes

```
      01001101
OR    11100100
      11101101
```

Figure 5.11 An example of the effect of OR

```
     00          01           10           11
OR   10     OR   10     OR    10     OR    10
     10          11           10           11
```

Figure 5.12

and colours that will draw/delete as required, as demonstrated in this program, which moves a triangle across the screen:

```
10 MODE 1
20 INK 3,24
30 DEFINT c,s,t,x,y
40 x=100:y=100
50 x1=100:y1=200
60 colour=1:colour1=24
70 type=3:shade=1
80 GOSUB 1000
90 GOSUB 2000
100 type=3:shade=2
110 WHILE x<639
120 x=x+4:x1=x1+4
130 GOSUB 2000
140 GOSUB 1000
150 x=x-4:x1=x1-4
160 GOSUB 2000:x=x+4:x1=x1+4
170 IF shade=2 THEN shade=1 ELSE shade=2
180 WEND
189 REM INKs back to normal
190 INK 1,24
200 INK 2,20
210 INK 3,6
220 END
998 REM swap INKs around: foreground bec
omes background
999 REM background becomes foreground
1000 IF colour=1 THEN colour=24:colour1=
1 ELSE colour=1:colour1=24
1010 INK 1,colour
1020 INK 2,colour1
1030 RETURN
```

```
1998 REM routine deletes/draws when tria
ngle is hidden
1999 REM (ie when it is in the backgroun
d colour)
2000 PRINT CHR$(23);CHR$(type);
2010 MOVE x,y
2020 DRAW x1,y1,shade
2030 DRAW x1+50,y1
2040 DRAW x+50,y1
2050 DRAW x,y
2060 IF type=2 THEN type=3 ELSE type=2
2070 RETURN
```

Line **100** temporarily moves the coordinates for the triangle back to the previous position, so that it can be erased while it is in the background colour. Line **120** switches between the colours needed to draw/delete, and **2030** switches between **OR** and **AND** line-drawing.

We can apply the technique to our original animation of the dog:

```
10 MODE 1
14 REM speed it up - use integers
15 DEFINT c,s,t,x,y
20 GOSUB 1000
30 GOSUB 2000
40 PRINT CHR$(23)CHR$(0);
50 END
998 REM read in 2 sets of 26 coordinates
999 REM for drawing 2 images of the dog
1000 DIM x(100),y(100)
1010 FOR count=1 TO 52
1020 READ x(count),y(count)
1030 NEXT
1040 RETURN
1050 DATA 0,0,20,40,20,40,10,80,10,80,0,
120,20,10,35,50,35,50,10,80,10,80,70,80
1060 DATA 60,0,80,40,80,40,70,80,80,10,9
5,50,95,50,70,80,70,80,90,140,90,140,110
,120,110,120,80,110
1070 DATA 5,10,25,40,25,40,15,80,15,80,1
0,120,0,15,30,50,30,50,15,80,15,80,75,80
1080 DATA 75,80,85,40,85,40,65,10,75,80,
90,50,90,50,60,15,75,80,100,130,100,130,
120,110,120,110,90,100
```

```
1999 REM set INK 3 to correct colour for EXOR
2000 INK 3,24
2010 colour=1:colour1=24
2020 type=3:shade=1
2029 REM draw figure at start position
2030 GOSUB 4000
2040 GOSUB 5000
2050 type=3:shade=2
2060 xinc=0
2070 WHILE x(1)+xinc<639
2079 REM update xinc to position for new figure
2080 xinc=xinc+20
2089 REM draw new figure in background colour
2090 GOSUB 5000
2099 REM switch INKs
2100 GOSUB 4000
2109 REM delete previous figure while in background colour
2110 xinc=xinc-20
2120 GOSUB 5000
2129 REM set xinc back to value for present figure
2130 xinc=xinc+20
2140 IF shade=2 THEN shade=1 ELSE shade=2
2150 WEND
2160 RETURN
2999 REM draw dog as a series of connecting points
3000 FOR count=start TO start+25 STEP 2
3010 MOVE x(count)+xinc,y(count)
3020 DRAW x(count+1)+xinc,y(count+1),shade
3030 NEXT
3040 RETURN
3999 REM switch INKs: foreground to background and vice versa
4000 IF colour=1 THEN colour=24:colour1=1 ELSE colour=1:colour1=24
4010 INK 1,colour
4020 INK 2,colour1
```

```
4030 RETURN
4999 REM vary graphics mode whether draw
ing or deleting
5000 PRINT CHR$(23)CHR$(type);
5010 IF type+shade=4 THEN start=1 ELSE s
tart=27
5020 GOSUB 3000
5030 IF type=2 THEN type=3 ELSE type=2
5040 RETURN
```

This gives a much smoother animation.

Exercises

1) Draw a series of circles in different positions in different **INK**s and produce a 'bouncing ball' animation by switching each circle between the foreground and background colours.
2) Create two alternative 'keep fit' images of a stick figure touching its arms to its sides and then putting them out straight.

Creating foreground and background colours

Let us now consider another consequence of this manipulation of **INK**s. This program draws two overlapping rectangles, but this time they are both colour-filled, one in yellow and the other in blue:

```
10 MODE 1
19 REM make yellow the 'foreground' colo
ur
20 INK 3,24
30 PRINT CHR$(23)CHR$(1);
40 x=100:y=100
50 xd=50:yd=100
60 INK 1,24
70 colour=1:GOSUB 1000
80 x=120:y=100
90 xd=100:yd=50
100 INK 2,20
110 colour=2:GOSUB 1000
```

```
120 PRINT CHR$(23)CHR$(0);
130 END
999 REM fill whole rectangle in
1000 FOR xcoord=x TO x+xd STEP 2
1010 MOVE xcoord,y
1020 DRAWR 0,yd,colour
1030 NEXT
1040 RETURN
```

If you run the program you will not be surprised to see that one rectangle obscures part of the second. What is perhaps surprising is that it is the yellow rectangle, which is drawn first, which hides part of the blue one, which is drawn second! This is the consequence of line **20**, where we have essentially said 'Make any overlapping area of the two colours appear as yellow'. We have decided that yellow will be the foreground colour and blue the background, and in any case where blue and yellow overlap, yellow takes priority.

If we change line **20** so that any overlapping area is displayed in blue, the blue rectangle now hides part of the yellow one:

```
19 REM make blue the 'foreground' colour
20 INK 3,20
```

This ability to give colours a priority is very useful in games. By suitable setting of the **INK**s we can arrange the colours so that a figure can pass over background features without erasing them. More impressively, the figure can pass BEHIND areas drawn in the foreground colour, emerging intact from the other side, as we shall now demonstrate.

In Chapter 2 we looked at one of the simplest ways of producing smooth animation, by using **TAG** and printing the character to its new position. The effect when there is a figure in the background is not unexpected:

```
10 MODE 1
20 x=230:y=130
29 REM draw and fill rectangle
30 FOR xcoord=x TO x+100 STEP 2
40 MOVE xcoord,y
50 DRAWR 0,100,1
60 NEXT
70 xprint=0:yprint=180
```

```
79 REM text or graphics cursor
80 TAG
89 REM print character to successive pos
itions
90 FOR xcoord=xprint TO 400 STEP 2
100 MOVE xcoord,yprint
110 PRINT CHR$(233);
120 NEXT
130 END
```

The character wipes out part of the background as it moves. Using a character with no 'border' on the left-hand side gives even worse results:

```
109 REM character with no left-hand bord
er: an arrow
110 PRINT CHR$(243);
```

The solution is to use **TAG** in combination with **EXOR** printing. However, simply printing the character is not enough:

```
10 MODE 1
20 x=230:y=130
29 REM draw and fill rectangle
30 FOR xcoord=x TO x+100 STEP 2
40 MOVE xcoord,y
50 DRAWR 0,100,1
60 NEXT
70 xprint=0:yprint=180
71 REM EXOR printing on
75 PRINT CHR$(23)CHR$(1);
80 TAG
89 REM print character to successive pos
itions
90 FOR xcoord=xprint TO 400 STEP 2
100 MOVE xcoord,yprint
110 PRINT CHR$(243);
120 NEXT
129 REM switch normal printing on
130 TAGOFF
140 PRINT CHR$(23)CHR$(0);
150 END
```

This is even more of a mess! The reason for this rather odd result is that the character is being **EXOR**ed with another version of itself printed one pixel to the right — the resulting combination after EXORing does not give the original character.

Let's devise our own arrow character with a border on the left, as in Figure 5.13. The arrow will be printed at the start position on-screen. Thereafter successive characters will be EXOR printed one pixel to the right of the present position. We must therefore define a second character which when EXORed with the original will result in a copy of the original displaced rightwards by one pixel. This is easier to understand in a diagram (see Figure 5.14).

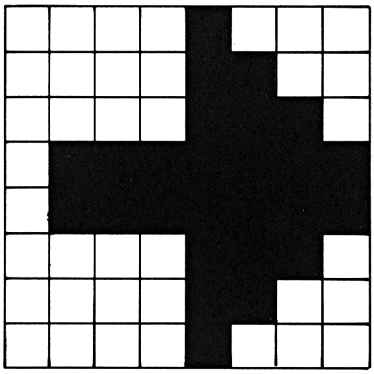


Figure 5.13 A character with a border on the left.

We can find the character definition for the EXOR arrow by examining the original arrow definition row by row and working out what value this needs to be EXORed with to reproduce the arrow. This is easier to do using binary, as can be seen in Figure 5.15, where we find the first number that will be needed in the EXOR character definition. So the first line of the EXOR character needs to be 24. We could continue in a similar vein, but in fact there is a quicker way of doing it. EXORing any binary number with a second gives a particular result. EXOR the original number with the result gives the second number back again. So we can find the new definition more quickly simply by EXORing every line of the original with the same line displaced one pixel rightwards, as in Figure 5.17. We can incorporate both the characters into a program to demonstrate that the EXOR approach successfully leaves the background intact:

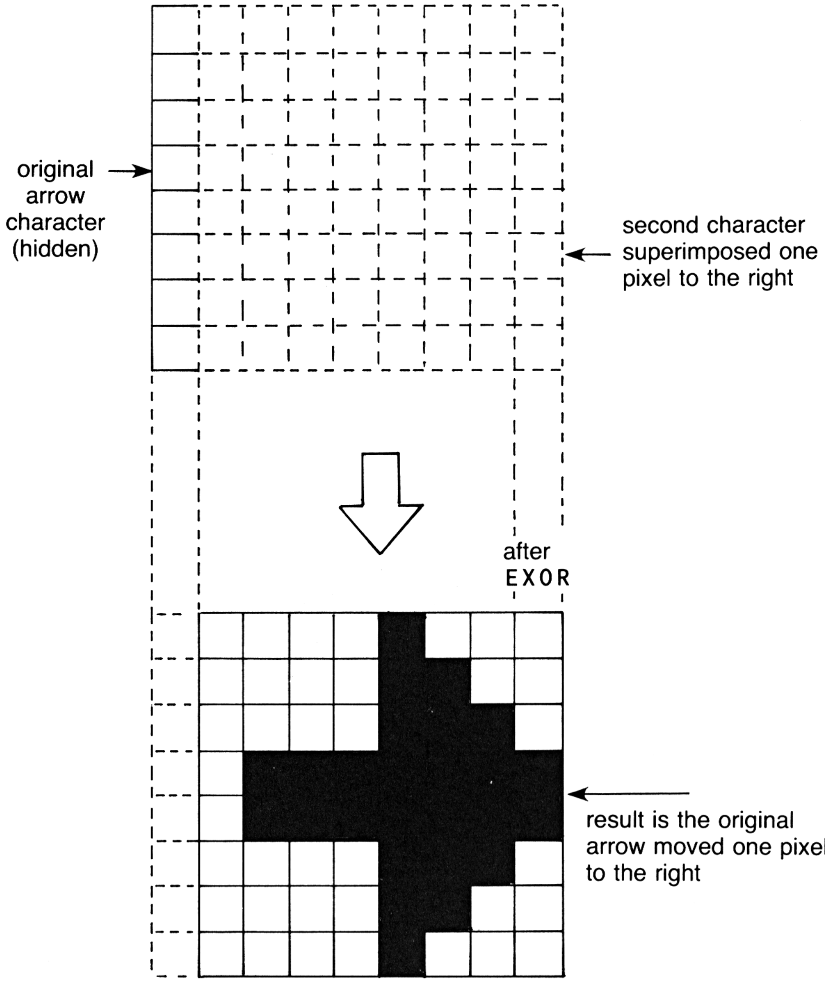


Figure 5.14 EXORing with a second character to create a copy of the first displaced one pixel to the right.

	00001000	Binary for first row of arrow character (8)
EXOR	00011000	needs to be EXORed with this binary number (24)
	00001000	to give the first code displaced to the right (8)

Figure 5.15

	00001000	Binary for first row of arrow character (8)
EXOR	00001000	result we want after EXORing (8)
	00011000	so we must EXOR with this number (24)

Figure 5.16

	00001100	Binary for second row of arrow character (12)
EXOR		
	00001100	result we want after EXORing (12)
	00010100	so we must EXOR with this number (20)
	00001110	Binary for third row of arrow character (14)
EXOR	00001110	
	00010010	result we want after EXORing (14)
		so we must EXOR with this number (18)

the same for the remaining rows of the character definition

Figure 5.17 Finding the character definition required to produce a copy of the arrow character displaced one pixel to the right following EXORing

```
84 REM define arrow character first prin
ted
85 SYMBOL 240,8,12,14,127,127,14,12,8
86 REM define EXOR arrow character
87 SYMBOL 241,24,20,18,129,129,18,20,24
90 FOR xcoord=xprint TO 400 STEP 2
100 MOVE xcoord,yprint
108 REM print normal arrow at first posi
tion
109 REM then EXOR arrow used to create a
rrow in new position
110 IF xcoord=xprint THEN PRINT CHR$(240
); ELSE PRINT CHR$(241);
120 NEXT
129 REM switch normal printing on
130 TAGOFF
140 PRINT CHR$(23)CHR$(0);
150 END
```

This opens the way to all sorts of impressive effects in games programs. By combining TAG, EXOR and changing the INKs, we can create a range of games characters that behave in different ways. Perhaps the player's 'man' can move across any yellow blocks on-screen, but not the blue ones; the fearsome 'ghosts' in pursuit can only cross the blue blocks; and the 'super-ghost' is not stopped by anything as it rushes in pursuit of the 'man'.

It is important to remember if you intend to create a character that it is still vital to have a one-pixel border, as the character will otherwise leave a trail when it moves. The border should be placed on the opposite side of the character to the direction in which movement will occur. In the arrow example a border was needed on the left because we were moving right.

If you intend to move a figure in more than one direction you will need to define an appropriate **EXOR** character for each direction of movement, and use that character for printing whenever a movement is made in that direction.

There is no need to confine yourself to single characters (although they are easier to handle). This program creates a three character 'car' and its three character **EXOR** equivalent. The car is moved across the screen, driving behind the yellow blocks and in front of the cyan blocks:

```
10 MODE 1
20 GOSUB 1000
30 GOSUB 2000
40 GOSUB 3000
50 TAGOFF
60 PRINT CHR$(23)CHR$(0);
70 END
998 REM overlap areas will be in INK 3,
which is set to yellow
999 REM this makes yellow the foreground
 colour
1000 INK 3,24
1010 y=100:colour=1
1020 FOR x=100 TO 500 STEP 50
1030 GOSUB 4000
1040 NEXT
1050 RETURN
2000 SYMBOL 240,0,15,28,124,127,18,12,0
2010 SYMBOL 241,0,255,120,120,255,255,0,
0
2020 SYMBOL 242,0,128,192,254,254,72,48,
0
2030 SYMBOL 243,0,16,36,132,128,55,20,0
2040 SYMBOL 244,0,0,137,137,0,1,0,0
2050 SYMBOL 245,0,128,64,2,2,216,80,0
2060 car$=CHR$(240)+CHR$(241)+CHR$(242)
2070 exorcar$=CHR$(243)+CHR$(244)+CHR$(2
45)
```

```
2080 RETURN
3000 PRINT CHR$(23)CHR$(1);
3010 TAG
3020 xprint=0:yprint=116
3030 FOR xcoord=xprint TO 550 STEP 2
3040 MOVE xcoord,yprint
3050 IF xcoord=xprint THEN PRINT car$; E
LSE PRINT exorcar$;
3055 r$="":WHILE r$="":r$=INKEY$:WEND
3060 NEXT
3070 RETURN
4000 IF colour=1 THEN colour=2 ELSE colo
ur=1
4010 FOR xcoord=x TO x+30 STEP 2
4020 MOVE xcoord,y
4030 DRAWR 0,100,colour
4040 NEXT
4050 RETURN
```

Line **1ØØØ** is necessary to set the overlap area of cyan car and yellow block to be yellow. Change the line to INK 3,6 (the normal colour for PEN 3) and you will see the car turn red as it passes the yellow block.

The effect is spoiled by the fact that the car reverts to the background colour when passing over the cyan blocks. This is because both colours being EXORed are cyan, and EXORing any binary number with itself always gives **Ø**, the current background colour.

	ØØ1Ø	A cyan point
EXOR	**ØØ1Ø**	being overwritten by another cyan point
	ØØØØ	gives a point in the background colour, blue

Figure 5.18

We could get around this problem by drawing the cyan blocks in a different colour and setting the right INK so that the overlap of car and block would not give this change in colour. The problem here is that we have only four colours in mode 1, and we are rapidly using up all the colours available! There is much more latitude in mode 0, where 16 different colours can be used, and we can redefine the PEN colours to give the desired effect.

```
10 MODE 0
998 REM INK 3 to yellow so cyan/yellow o
verlap gives yellow
999 REM so yellow is foreground colour
1000 INK 3,24
1001 REM INK 6 to cyan so cyan/white ove
rlap gives cyan
1002 REM so white is background colour
1003 INK 6,20
2071 MOVE 0,0
2072 DRAW 0,0,2
4000 IF colour=1 THEN colour=4 ELSE colo
ur=1
```

In mode 0 each colour is represented by four bits. The car is **EXOR** printed using **PEN 2**, and this can overlap either with a yellow block (drawn with **PEN 1**) or a red block (drawn with **PEN 4**). We therefore need to reset **PEN 3** to show yellow (so that the car appears to go behind the yellow blocks) and **PEN 6** to show cyan (so that the car goes in front of the red blocks), lines **1000,1001**. You can reverse the priority so that the car goes in front of the yellow blocks and behind the cyan by putting:

```
998 REM INK 3 to cyan so cyan/yellow ove
rlap gives cyan
999 REM so yellow is background colour
1000 INK 3,20
1001 REM INK 6 to white so cyan/white ov
erlap gives white
1002 REM so white is foreground colour
1003 INK 6,26
```

	0010	The cyan car
EXOR	0001	overlapping a yellow block
	0011	produces the code for PEN 3, red
	0010	The cyan car
EXOR	0100	overlapping a white block
	0110	produces the code for PEN 6, cyan

Figure 5.19

0000	Blue	–	far background colour
0001	Yellow	–	near background colour
0010	Cyan	–	midground colour
0011	Cyan	–	midground hiding near background
0100	White	–	foreground colour
0101	White	–	foreground hiding near background
0111	White	–	foreground hiding midground hiding background

Figure 5.20 Creating background, midground and foreground colours by setting appropriate INKs

One slight complication here is that TAG causes printing to take place using the current graphics foreground colour. We can't use a PEN command to change that colour any more, because after a TAG the computer changes the colour only when obeying a graphics command.

Before the car can be printed in cyan, the graphics colour must be switched from white to cyan, and to do this we need to issue a graphics instruction. This is the reason for lines 2071 and 2072 — without them the car is printed using the current graphics colour, white.

Mode 0 offers plenty of scope for extending the foreground/background idea, because of the range of colours available. We can, for example, set up background, midground and foreground colours, as in Figure 5.20. Here the fourth bit set to 1 indicates the presence of yellow, the near background colour; the third bit set to 1 shows the presence of cyan, the midground colour; and the second bit set to 1 indicates white, the foreground colour. Other combinations of bits show one colour obscuring another.

By EXOR drawing, we can draw or delete lines in any of the colours without disturbing overlapping or hidden lines in other colours. For example, let's see the effect of EXOR on a white foreground line hiding a cyan midground line which in turn hides a yellow background line. Let's first delete the white line (see Figure 5.21). What if we had deleted the cyan midground line? (see Figure 5.22.)

	0111	White point hiding cyan hiding yellow
EXOR	0100	on deletion of the white foreground
	0011	the cyan midground is revealed

Figure 5.21

	0111	White point hiding cyan hiding yellow
EXOR	0010	on deletion of the cyan midground point
	0101	the white foreground now only hides the yellow background

Figure 5.22

By examining the results of deleting one line from other combinations we can determine the **INK** commands we need to use to allow deletion of any of the lines without apparent disturbance to the others. This can be seen in the following program, which draws and fills in a white foreground rectangle on a cyan midground rectangle, resting in turn on a yellow background rectangle:

```
10 MODE 0
20 GOSUB 1000
30 GOSUB 2000
40 GOSUB 3000
50 PRINT CHR$(23)CHR$(0);
60 MODE 1
70 END
999 REM set PENs to produce appropriate
INKs
1000 INK 3,20
1010 INK 5,26
1020 INK 6,26
1030 INK 7,26
1040 RETURN
1999 REM draw 3 rectangles on top of eac
h other
2000 xyellow=100:yyellow=40:sideyellow=3
00
2010 xcyan=150:ycyan=80:sidecyan=220
2020 xwhite=200:ywhite=120:sidewhite=140
2029 REM set EXOR graphics
2030 PRINT CHR$(23)CHR$(1);
2040 colour=1:x=xyellow:y=yyellow:side=s
ideyellow
2050 GOSUB 4000
2060 colour=2:x=xcyan:y=ycyan:side=sidec
yan
2070 GOSUB 4000
2080 colour=4:x=xwhite:y=ywhite:side=sid
ewhite
```

```
2090 GOSUB 4000
2100 RETURN
3000 WINDOW 1,20,25,25
3009 REM input commands
3010 WHILE response$<>"e"
3020 INPUT"Command (y/c/w/e)",response$
3029 REM draw or delete yellow rectangle
3030 IF response$="y" THEN colour=1:x=xy
ellow:y=yyellow:side=sideyellow:GOSUB 40
00
3039 REM draw or delete cyan rectangle
3040 IF response$="c" THEN colour=2:x=xc
yan:y=ycyan:side=sidecyan:GOSUB 4000
3049 REM draw or delete white rectangle
3050 IF response$="w" THEN colour=4:x=xw
hite:y=ywhite:side=sidewhite:GOSUB 4000
3890 WEND
3900 RETURN
4000 FOR xcoord=x TO x+side STEP 4
4010 MOVE xcoord,y
4020 DRAWR 0,side,colour
4030 NEXT
4040 RETURN
```

Line 3020 enables you to draw/delete the yellow, cyan or white rectangle by inputting the letter **y**, **c** or **w** respectively (**e** to end). The effects in this program may surprise you — you can delete or draw the yellow background rectangle and leave the two rectangles that rest upon it completely untouched!

Exercises

1) Define a 'boat' using one or more characters, as well as an appropriate EXOR character so that it can be moved smoothly across the screen.
2) Extend the previous program by sailing the 'boat' across some blue water, where it travels in front of a number of large and menacing reefs that push up out of the water.
3) Add some grassy hills which the boat sails behind.
4) Sail another boat in the opposite direction. When the boats meet, one passes behind the other.
5) Change the order of priority of the colours in the rectangle program so that yellow becomes the foreground colour and

white the background. Only the yellow rectangle should be visible at the start, and the white one should only appear after the other two have been erased.

6) Add a fourth rectangle in a colour which becomes the new foreground, white now being a fore-midground colour and cyan a back-midground colour. Adjust the INKs so that any of the rectangles can be drawn or erased without affecting the others.

Chapter Six

. . . and artistry

In Chapter 4 we developed a program that enabled the user to draw on the screen and save the resulting picture in a file for future use. In this chapter we shall develop the program further and extend its facilities, so that standard shapes can be drawn and coloured in.

Selecting options from the MENU

The earlier program relied entirely on keyboard input. Most programs of this type operate by displaying a MENU on-screen. This is a display showing the range of options currently available to the user. Typical options might be to rescale the drawing, select a new colour for shading, or draw a circle or rectangle (see Figure 6.1).

Selections are made from the menu by moving the cursor to the appropriate position on the screen. The computer notes the position of the cursor and implements the choice indicated by that position. This approach makes the program a lot easier to use if the menu meanings are obvious: it is no longer necessary to remember which key on the keyboard changes the line colour, which indicates you wish to draw a circle, and so on.

However, unless an alternative input device such as a joystick is available, some controls must still be operated by key depression: the most obvious being the movement of the cursor and the fixing of a point. Additionally we shall leave the

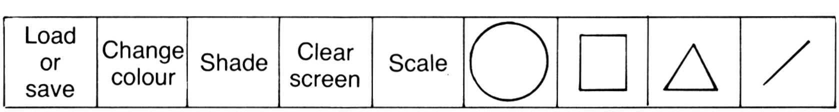


Figure 6.1 A typical menu for a drawing program.

line-on/line-off choice as a keyboard control — it would be a tedious business to have to move to the menu every time this was changed.

Our new program will include options for selecting line colour (any of eight colours, including the background colour) and choosing to draw either single lines, a circle, a triangle or a rectangle. A further menu option will be a 'toggle' between colour-fill and no colour-fill, so that the user can either shade a closed figure in the current foreground colour or leave it empty.

Selection of a choice from the menu is shown by pressing the '**f**' (fix point) key once the cursor is in the right place. The computer will beep to acknowledge the choice. It can be a bit difficult to remember whether the circle-drawing mode is in operation or what the current foreground colour is, so we will provide a reminder. This will take the form of a character at the bottom left corner of the screen, which will always reflect the current situation. The menu itself will also be shown along the bottom of the screen, although its positioning is a matter of personal taste — you may prefer to run it down the left-hand side or along the top.

As before, we begin with a basic program:

```
10 MODE 0
20 GOSUB 1000
30 GOSUB 2000
40 END
1000 startx=320:starty=200
1010 x=startx:y=starty
1020 foregroundcolour=1
1030 linedraw=0
1037 REM print symbol at bottom left to
show
1038 REM current situation - info$ start
s as yellow line
1039 REM because at start INK used is ye
llow and cursor draws lines
1040 info$=CHR$(47)
1050 LOCATE 1,24
1060 PRINT info$;
1070 menux=5:menuy=24
1079 REM define triangle, and colour-fil
l/no-colour-fill symbols
```

```
1080 SYMBOL 240,0,2,6,10,18,34,66,254
1090 SYMBOL 241,255,129,129,129,129,129,
129,255
1099 REM print samples of colours from P
ENs 0 to 7
1100 LOCATE menux,menuy
1110 PRINT CHR$(241);
1120 FOR colour=1 TO 7
1130 PEN colour
1140 PRINT CHR$(143);
1150 NEXT
1160 PEN 1
1169 REM print symbols for line, circle,
 rectangle, triangle, no-colour-fill
1170 PRINT CHR$(47)CHR$(79)CHR$(232)CHR$
(240)CHR$(241);
1180 PRINT CHR$(23)CHR$(1);
1900 RETURN
1999 REM plot cursor at start position
2000 GOSUB 3000
2009 REM repeat until 'e' pressed
2010 WHILE response$<>"e"
2019 REM delete cursor via EXOR
2020 GOSUB 3000
2029 REM scan keyboard for input
2030 GOSUB 4000
2039 REM draw cursor
2040 GOSUB 3000
2900 WEND
2910 RETURN
2999 REM routine to draw/delete lines
3000 PLOT x,y,foregroundcolour
3010 IF linedraw=0 THEN RETURN
3020 DRAW startx,starty
3030 RETURN
3999 REM routine toscan keyboard for inp
ut and implement choice
4000 response$=LOWER$(INKEY$)
4010 IF response$="a" THEN y=y+2
4020 IF response$="z" THEN y=y-2
4030 IF response$="," THEN x=x-4
4040 IF response$="." THEN x=x+4
4048 REM Space Bar depression fixes poin
t
```

```
4049 REM if y coordinate high enough on
screen
4050 IF response$=" " AND y>31 THEN GOSU
B 5000:linedraw=foregroundcolour
4059 REM otherwise Space Bar indicates m
en choice
4060 IF response$=" " AND y<32 THEN GOSU
B 6000
4069 REM line draw on/off toggle
4070 IF response$="1" THEN IF linedraw=0
 THEN linedraw=foregroundcolour ELSE lin
edraw=0
4900 RETURN
4999 REM permanent fixing of line
5000 PRINT CHR$(23)CHR$(0);
5010 GOSUB 3000
5020 PRINT CHR$(23)CHR$(1);
5030 startx=x:starty=y
5900 RETURN
5999 REM menu choice - reject if not on
menu
6000 IF x<128 OR x>543 OR y<16 THEN RETU
RN
6010 SOUND 7,400
6019 REM x coord<384 shows a colour chan
ge is needed
6020 IF x<384 THEN foregroundcolour=TEST
(x,y):GOSUB 7000:RETURN
6029 REM check x coordinate to deduce ch
oice
6030 IF x<416 THEN info$=CHR$(47):GOSUB
7000:RETURN
6040 IF x<448 THEN info$=CHR$(79):GOSUB
7000:RETURN
6050 IF x<480 THEN info$=CHR$(232):GOSUB
 7000:RETURN
6060 IF x<512 THEN info$=CHR$(240):GOSUB
 7000:RETURN
6069 REM must be colour-fill/no-colour-f
ill toggle
6070 GOSUB 15000
6080 RETURN
7000 PEN foregroundcolour
7010 LOCATE 1,24
```

```
7020 PRINT info$
7030 RETURN
15000 REM coming up soon!
15010 RETURN
```

This enables us to move the cursor and to make choices from the menu — although at present only the colour change actually works!

A menu choice is indicated if the y coordinate of the cursor is less than 32 when a point is fixed. Lines `4050` and `4060` could be combined into a single `IF . . . THEN . . . ELSE` statement, but are kept separate for clarity. Subroutine `6000` checks that the x coordinate is within the menu area, and then implements the choice depending on that coordinate. Subroutine `7000` prints the character `info$` to the bottom left corner of the screen: this string variable is changed whenever a new menu selection is made, so that it is always the right colour and symbol.

Adding the standard shapes is relatively straightforward, but we must think carefully about how these choices are implemented. Circle-drawing is not very fast, as we saw in Chapter 3, and so it would be unwise to continually draw/delete a potential circle using `EXOR`, as the program would be unacceptably slow. Once the circle option is selected, it operates as follows: the first point fixed is taken to be the centre of the circle; the point next fixed is taken as a point on the circumference of the desired circle; the circle is drawn with a radius equal to the distance between the two points.

`EXOR` drawing of a triangle or rectangle is feasible as only a few lines are involved, but the two options need to work slightly differently. Once the first point on a triangle has been fixed, we still have no idea where the remaining points will be, but this is not the case with the rectangle. Once the first corner has been fixed, the position of the remaining corners depend entirely on where the diagonally opposite corner is placed. The rectangle option therefore involves the fixing of just two points. The first point is taken to be one corner of the rectangle, and the second that of the corner diagonally opposite.

Both the triangle and rectangle drawing options involve the movement of the cursor, but whereas previously `EXOR`

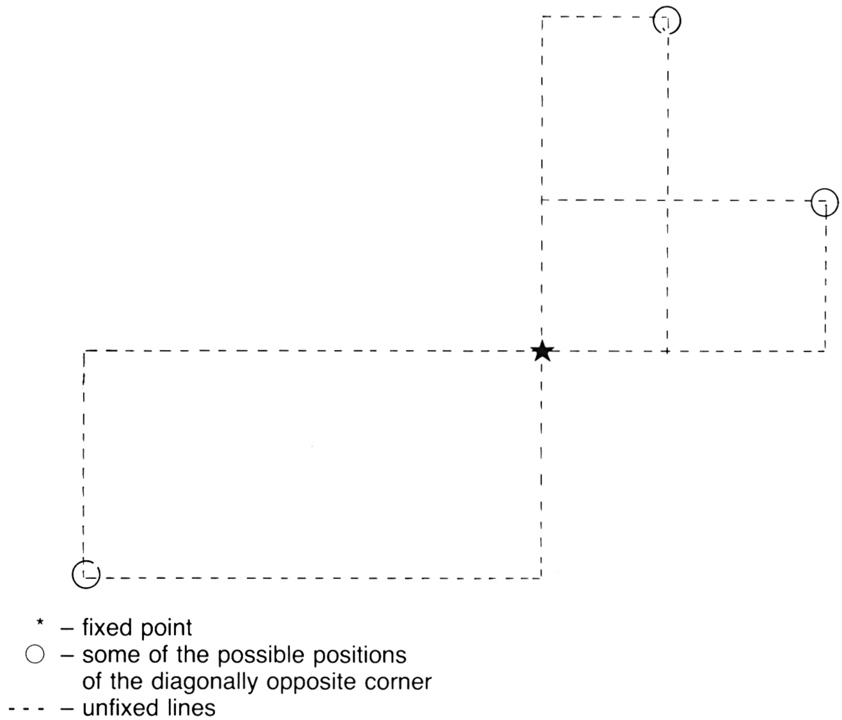


Figure 6.2 Fixing the position of a rectangle by identifying just two corners.

drawing has only involved a single line, here we need to draw and delete a number of lines simultaneously. These options would therefore need to be entirely separate subroutines that call the censor movement subroutine but not the line draw/delete subroutine used the rest of the time. We shall settle for a simpler approach where selection of the triangle or rectangle option results in the shape being drawn only once an appropriate number of points have been fixed. This is less attractive than the constant display of the (potential) triangle or rectangle by **EXOR** drawing, but this is left as an exercise for the reader.

```
4999 REM check for circle/rectangle/tria
ngle menu choice
5000 PRINT CHR$(23)CHR$(0);
5001 IF circle>0 THEN GOSUB 8000
5002 IF rectangle>0 THEN GOSUB 9000
5003 IF triangle>0 THEN GOSUB 10000
5010 GOSUB 3000
```

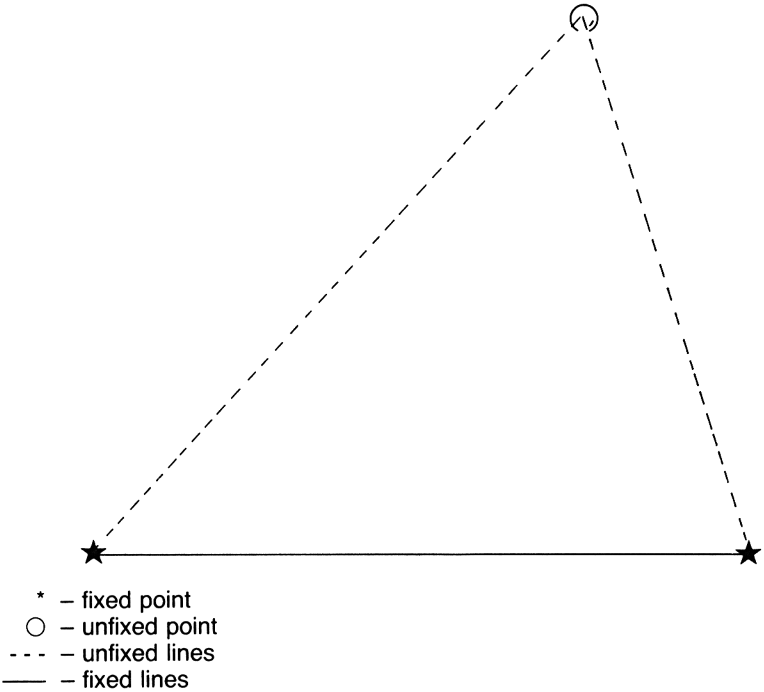


Figure 6.3 Before the third corner of a triangle is fixed, two of its sides could be in any position.

```
5020 PRINT CHR$(23)CHR$(1);
5030 startx=x:starty=y
5900 RETURN
5999 REM menu choice - reject if not on
menu
6000 IF x<128 OR x>543 OR y<16 THEN RETU
RN
6010 SOUND 7,400
6019 REM x coord<384 shows a colour chan
ge is needed
6020 IF x<384 THEN foregroundcolour=TEST
(x,y):GOSUB 7000:RETURN
6029 REM check x coordinate to deduce ch
oice
6030 IF x<416 THEN info$=CHR$(47):GOSUB
7000:RETURN
6040 IF x<448 THEN info$=CHR$(79):GOSUB
7000:circle=1:RETURN
6050 IF x<480 THEN info$=CHR$(232):GOSUB
```

```
 7000:rectangle=1:RETURN
6060 IF x<512 THEN info$=CHR$(240):GOSUB
 7000:triangle=1:RETURN
6069 REM must be colour-fill/no-colour-f
ill toggle
6070 GOSUB 15000
6080 RETURN
7000 PEN foregroundcolour
7010 LOCATE 1,24
7020 PRINT info$
7025 IF x<384 THEN RETURN
7029 REM new menu choice - cancel previo
us choice
7030 circle=0
7040 triangle=0
7050 rectangle=0
7060 RETURN
7999 REM circle-drawing routine: centre
and point on circumference required
8000 IF circle=1 THEN circle=2:RETURN
8009 REM if we get here we have 2 points
 needed and can find radius
8010 xd=ABS(x-startx):yd=ABS(y-starty)
8020 radius=SQR(xd*xd+yd*yd)
8030 MOVE startx,starty+radius
8040 FOR angle=0 TO 2*PI STEP PI/60
8050 DRAW startx+radius*SIN(angle),start
y+radius*COS(angle)
8060 NEXT
8070 DRAW startx,starty+radius
8080 PLOT startx,starty,0
8089 REM set flag back to 1 for next cir
cle
8090 circle=1
8100 RETURN
8999 REM rectangle-drawing routine: two
corners required
9000 IF rectangle=1 THEN rectangle=2:RET
URN
9010 MOVE startx,starty
9020 DRAWR x-startx,0,foregroundcolour
9030 DRAWR 0,y-starty
9040 DRAWR startx-x,0
9050 DRAWR 0,starty-y
```

```
9059 REM set flag back to 1 for next rec
tangle
9060 rectangle=1
9070 RETURN
9999 REM triangle-drawing routine: three
 points required
10000 IF triangle=1 THEN triangle=2:x1=x
:y1=y:RETURN
10010 IF triangle=2 THEN triangle=3:RETU
RN
10020 MOVE x,y
10030 DRAW startx,starty,foregroundcolou
r
10040 DRAW x1,y1
10050 DRAW x,y
10059 REM set flag back to 1 for next tr
iangle
10060 triangle=1
10070 RETURN
15000 REM coming up soon!
15010 RETURN
```

The selection of the circle-, rectangle- or triangle-drawing option is indicated by setting a flag to 1. Subroutine **7000** is extended so that selecting any option sets the flag for the others to **0** — otherwise the computer might try to draw a triangle AND rectangle, for example! Line **7025** is included because we don't want to reset the flags if the menu choice has only involved a change of colour.

As the three figure-drawing options will only be implemented once the right number of points have been fixed, the check for these options is placed within the point-fixing routine, subroutine **5000**. If the flag for a particular option is greater than zero, the appropriate subroutine is called, lines **5001–5003**.

The figure-drawing subroutines at **8000**, **9000** and **10000** all have one thing in common. If the number of points fixed since the option was selected is not yet great enough, the flag is increased in value by 1 to indicate another point fixed, and the subroutine then ends, lines **8000**, **9000**, **10000** and **10010**. Effectively the flag is also used as a counter to show how many points have been fixed. Two points must be fixed before the circle or rectangle options can work, and three points are needed before the triangle option can operate.

Once the right number of points has been fixed the subroutine draws the figure and sets the flag back to 1. This means that once, for example, the circle-drawing option has been chosen, circle will continue to be drawn until another option is chosen from the menu. Don't forget this — it's very easy to try to draw a line while in rectangle or triangle mode, with unexpected results!

The structure of the program means that you could use the same approach to add further figure-drawing options to the menu, to make it easy to draw ellipses, diamonds, etc.

Exercises

1) Extend the range of colours displayed by the menu to 10 of your own choosing.
2) Introduce a 'clear screen' command onto the menu. This clears the entire graphics area to the current foreground colour.
3) Add a new figure-drawing option which allows the drawing of arcs. You will need to specify three points: the centre of the circle of which the arc is part, and the start and end of the arc.

Colour it in

We now come to the colour-fill routine. The Amstrad does have a command that enables large areas to be filled with a colour, the **WINDOW** command, but unfortunately this is related to text coordinates only. We must therefore devise our own method for colouring a graphics area.

It is clear that to be completely sure of filling a closed figure with colour we will have to examine the state of every point within the figure. We will tackle the problem in stages, first devising a routine that will colour every point on a line and then extending it.

Suppose we choose an arbitrary point within our figure. We can colour all points that lie on the same line in two stages. First, the point to the left of the present position is examined

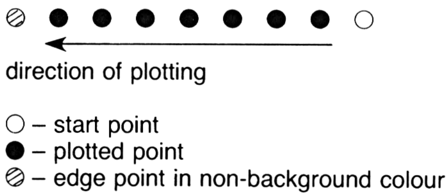


Figure 6.4

using **TEST**. If it is in the background colour, the point is plotted in the present foreground colour. This point becomes the new 'start' point, and the point to its left is examined. The process is repeated until a point in a non-background colour is encountered — this must be the edge of the figure (see Figure 6.4).

Effectively a line has been drawn from the initial point leftwards to the boundary of the shape. The remaining points on the line can be coloured by repeating the whole procedure from the start point, only this time points to the right are successively examined. The following routine carries out the line-filling using the procedure described:

```
1 REM this is a program in its own right
2 REM although the subroutines etc will
3 REM be incorporated into our main prog
ram
10 MODE 0
19 REM draw your own shape here - this i
s a triangle
20 MOVE 200,100
30 DRAW 450,350
40 DRAW 340,400
50 DRAW 200,100
54 REM choose 10 random points and plot
lines from them
55 FOR count=1 TO 10
59 REM random point within the triangle
60 rand=INT(RND(1)*230):xhere=210+rand:y
here=110+rand
70 foregroundcolour=1:yfill=yhere
79 REM fill points to left of the point
80 xinc=-4:xfill=xhere
90 GOSUB 10000
98 REM now points to right - don't forge
t start point itself
```

```
99 REM so begin one place to left
100 xinc=4:xfill=xhere-4
110 GOSUB 18000
120 NEXT
130 END
17999 REM check point by point to the le
ft or right
18000 t=0:WHILE t=0
18010 xfill=xfill+xinc
18020 t=TEST(xfill,yfill)
18028 REM if t=0 point is in background
colour
18029 REM and so must be plotted
18030 IF t=0 THEN PLOT xfill,yfill,foreg
roundcolour
18040 WEND
18050 RETURN
```

Subroutine **18000** is called twice with different increments to the x coordinate: initially the increment is −4 (examining points to the left) and then +4 (examining points to the right). (Note that this increment will vary with the mode — the resolution in mode 1 is higher and an increment of +/−2 would be required.)

You can confirm for yourself that the routine always works by drawing a different shape at the start and setting x here and y here to any coordinates within the figure.

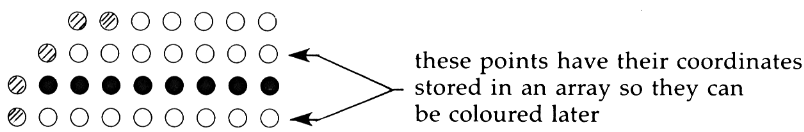
What happens if the point lies outside the figure? Unfortunately the **TEST** command is of little help here, and one of the Amstrad's other capabilities works against us. If the initial point is not within a closed figure, the Amstrad will carry on examining points to the left, even when they go off-screen! The **TEST** command regards a point off-screen as being in the background colour, and the computer, treating its examination of points as unfinished, will carry on taking 4 from the x coordinate even if the x coordinate has become −1000!

There are several ways around this. We can include a test on the value of xfill:

```
59 REM random point deliberately outside
 the triangle
18000 t=0:WHILE t=0 AND xfill>0 AND xfil
l<639
```

This obviously slows the program down, as the x coordinate for every point must be tested. An alternative approach which is faster is to draw a 'frame' around the edge of the screen. The computer will stop plotting points when it reaches the edge of the screen as the next point is in a non-background colour.

Our routine for filling in single lines is now complete. How can we extend it so that it fills a figure? One way is to examine the points immediately above and below the line we have just filled. If a point is in the background colour, its coordinates can be stored in an array to make sure it serves as the starting point for another line drawn later (see Figure 6.5).



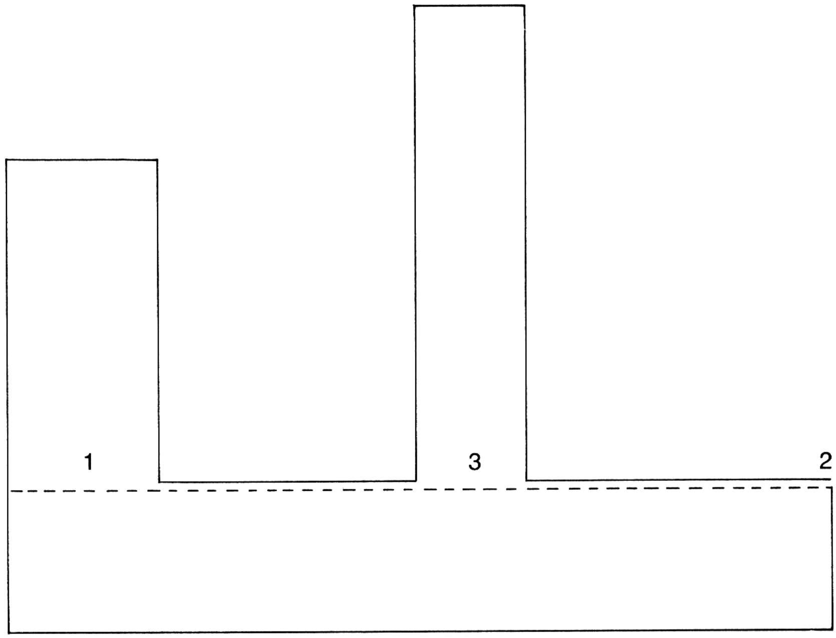
○ – points in background colour
● – plotted point
⊘ – edge point in non-background colour

Figure 6.5

It might seem that we need only continue testing points until we find one in the foreground colour, but some shapes require the testing of every point, as in Figure 6.6. Only some of the vertical parts of the figure would be coloured unless every point above the line was checked. The checking of every point is rather time-consuming, and we here have to make a choice between efficiency and perfection. Do we want a fast colour-fill routine which sometimes fails, or a slower routine which will always colour the points within a shape, no matter how convoluted it is?

An imperfect routine which nonetheless successfully fills many shapes involves the checking of only 4 points:

```
10 MODE 0
14 REM array to hold coordinates of non-
coloured points
15 DIM x(300),y(300)
19 REM draw your own shape here
20 startx=150:starty=150
30 MOVE startx,starty
40 DRAWR 100,0
```



- - - most recently plotted line
1 – points just above the line here will be coloured
2 – points just above the line here are already coloured, so checking will go no further
3 – points here are uncoloured and will remain so unless every point above the plotted line is checked

Figure 6.6

```
50 DRAWR 0,100
60 DRAWR -100,0
70 DRAWR 0,-100
78 REM x and y are coordinates of a poin
t
79 REM within the shape
80 x=200:y=200
90 foregroundcolour=1
100 GOSUB 15000
110 PRINT CHR$(23)CHR$(0);
999 END
15000 begin=2:finish=1
15010 xfill=x:yfill=y
15020 x(2)=x:y(2)=y
15030 PRINT CHR$(23)CHR$(0);
15040 PLOT x,y,0
```

```
15050 GOSUB 16000
15060 GOSUB 17000
15070 PRINT CHR$(23)CHR$(1);
15080 IF circle>0 OR rectangle>0 OR tria
ngle>0 THEN PLOT x,y,foregroundcolour
15090 RETURN
15999 REM check colour of present point
16000 xhere=xfill:yhere=yfill
16010 IF TEST(xfill,yfill)<>0 THEN RETUR
N
16020 xinc=-4
16029 REM check colour of all points to
tis left
16030 GOSUB 18000
16040 xfill=xhere-4:yfill=yhere
16049 REM check colour of all points to
tis right
16050 xinc=4
16060 GOSUB 18000
16070 RETURN
16999 REM keep checking points in list u
ntil they're all done
17000 WHILE begin<>(finish+1)MOD 300
17010 xfill=x(begin):yfill=y(begin)
17020 begin=(begin+1)MOD 300
17030 GOSUB 16000
17040 WEND
17050 RETURN
17999 REM check point by point to the le
ft or right
18000 t=0:WHILE t=0
18010 xfill=xfill+xinc
18020 t=TEST(xfill,yfill)
18030 IF t=0 THEN PLOT xfill,yfill,foreg
roundcolour
18040 WEND
18050 xfill=xfill-xinc:yfill=yfill-2
18059 REM if point on line below is not
coloured, save it
18060 IF TEST(xfill,yfill)=0 THEN GOSUB
19000
18070 yfill=yfill+4
18079 REM if point on line above is not
coloured, save it
```

```
18080 IF TEST(xfill,yfill)=0 THEN GOSUB
19000
18090 RETURN
18999 REM store coordinates of uncoloure
d point so it can be coloured later
19000 finish=(finish+1)MOD 300
19010 x(finish)=xfill:y(finish)=yfill
19020 RETURN
```

Try running the program with a variety of different shapes. It only examines points diagonally above and below the ends of the present line. As a number of lines will be filled, the coordinates of their ends are saved in the arrays x(), y(). Two pointers, 'begin' and 'finish' indicate positions in the array; 'begin' gives the number of the array elements containing the (x,y) coordinate of the next point to be examined, and 'finish' gives the number of the next 'free' array elements in which the coordinates of new non-coloured points can be stored.

For example, after the first line has been drawn, it is quite likely that the four points above and below the line will be found to be in the background colour, and their coordinates will therefore be stored in the arrays x() and y(). Subroutine **17000** successively examines each point whose coordinates are in the array, and subroutine **16000** sets up the examination of points on either side of this. The points are coloured in subroutine **18000**, until a point not in the background colour is found, line **18030**, when the search ends. Lines **18050** to **18080** then check the colour of the points diagonally above and below the end-point, and subroutine **19000** stores them if it turns out they are in the background colour.

Lines **17000** and **19000** contain a **MOD** because the arrays are treated as circular. Obviously we cannot know how many coordinates the computer needs to store, and 300 seems a

⊘ – edge points
● – plotted points making up the line
* – points whose colour is checked

Figure 6.7

reasonable number. A large and complicated figure might require more, however, but we can avoid having to set up an even larger array by re-using the earlier elements. This will not result in any loss of data: by the time we need to use the lower array elements again they will have been examined and are no longer needed.

The routine fills 90% of a circle, but fails towards the top and bottom where the arc contains short horizontal lines. These are examined and found not to be background points, and the program abandons the colour-fill because it appears to have reached a boundary line. We can avoid this problem with very little deterioration in speed simply by examining two more points:

```
19 REM rectangle with a vertical 'arm'
20 startx=150:starty=150
30 MOVE startx,starty
40 DRAWR 100,0
50 DRAWR 0,100
55 DRAWR -10,0
56 DRAWR 0,50
57 DRAWR -60,0
58 DRAWR 0,-50
60 DRAWR -30,0
70 DRAWR 0,-100
16000 xhere=xfill:yhere=yfill
16010 IF TEST(xfill,yfill)<>0 THEN RETUR
N
16020 xinc=-4
16029 REM check colour of all points to
tis left
16030 GOSUB 18000
16031 REM keep coordinates of left end o
f line
16032 leftx=xfill
16040 xfill=xhere-4:yfill=yhere
16049 REM check colour of all points to
tis right
16050 xinc=4
16060 GOSUB 18000
16061 REM keep coordinates of right end
of line
16062 rightx=xfill
```

```
16063 REM find mid-point of line and che
ck
16064 REM points above and below mid-poi
nt as well
16065 xfill=(leftx+rightx)/2:yfill=yhere
-2
16066 IF TEST(xfill,yfill)=0 THEN GOSUB
19000
16067 yfill=yhere+2
16068 IF TEST(xfill,yfill)=0 THEN GOSUB
19000
16069 REM this isn't perfect, but it's b
etter!
16070 RETURN
```

This successfully fills circles, and only fails on shapes with vertical branches which join the main region at a horizontal line. The position of the initial point makes a difference as to how much of the region is coloured. If any failure occurs the remaining area can be filled by using a new start point, so the routine seems to offer a good compromise between speed and perfection. It can be incorporated into the main program as follows:

```
15 DIM x(300),y(300)
5031 IF fill=1 AND circle=0 AND rectangl
e=0 AND triangle=0 THEN GOSUB 15000
6069 REM must be colour-fill/no-colour-f
ill toggle
6070 IF fill=1 THEN fill=0:fill$=CHR$(24
1) ELSE fill=1:fill$=CHR$(233)
6080 LOCATE 17,24
6090 PRINT fill$;
6100 RETURN
8091 IF fill=1 THEN x=startx:y=starty:GO
SUB 15000
9061 IF fill=1 THEN x=(x+startx)/2:y=(y+
starty)/2:GOSUB 15000
10061 IF fill=1 THEN x=(x+startx+x1)/3:y
=(y+starty+y1)/3:GOSUB 15000
10071 REM then add the fill routine from
 line 15000
10072 REM to line 19020
```

Colour-fill is implemented beginning at any point fixed after the option has been chosen. Colour-fill may be operating at the same time as the circle-, rectangle-, or triangle-drawing options, in which case the figure is drawn and automatically filled in by the computer choosing a start point within the figure, lines 8091, 9061, and 10061. The other possibility is that colour-fill is being used to fill some other shape, and this is catered for in 5031. Note that you must switch colour-fill off if you wish to be able to draw lines — the program will otherwise interpret the fixing of a point as the position for start of colour-fill and try to fill the screen with colour!

Exercises

1) As individual points are plotted in the colour-fill routine, it is easy to use combinations of colour to fill a figure by, for example, toggling between different foreground colours at line 18030. Add a menu option to allow colour-fill with a mixture of any two colours selected from the rest of the menu.
2) A flaw in the present program is that colour-fill stops as soon as any point not in the background colour is encountered. This would make it impossible to colour-fill any shape that has been drawn on an area already colour-filled — a blue door could not be coloured against the background of a red house, for example. Modify the program so that colour-fill fills any figure, no matter what other colours are present.
3) Extend the colour-fill algorithm in the direction suggested, so that it will work perfectly for any shape.

Save the masterpiece

Now that we can create a colourful picture it would be nice to save it. In Chapter 4 we stored details of the coordinates of points and lines in several arrays which were then saved as a file. The same approach could be applied here although the program would require some modification: we would need to know if points were used in the circle- or rectangle-drawing options, and we would also have to note if a figure was colour-filled or not.

An alternative is to save a copy of the entire screen display instead. When we want to view or extend the latest masterpiece, it can be loaded back onto the screen and we can carry on drawing using the usual menu options. This method has the advantage that it can be used with the program as it stands.

Saving the screen is only a particular example of the facility that the Amstrad has for saving a copy of a section of computer memory. As we have noted previously, the screen display is a representation of part of the computer RAM. By saving the appropriate memory locations we effectively store a copy of the screen on tape.

The Amstrad needs some information to save a copy of the memory: where the section of memory starts and how long (in terms of bytes) the section is. This information is also saved, and so there is no need to provide it when loading back into the computer.

The routine for loading and saving are easily incorporated into the program, being called from the keyboard by depression of '**i**' (for input) or '**o**' (for output) respectively:

```
4080 IF response$="i" THEN GOSUB 11000
4090 IF response$="o" THEN GOSUB 12000
10999 REM set up window so that picture
is not disturbed
11000 WINDOW 1,20,24,25:PEN 1
11010 PRINT "To load a picture"
11020 INPUT "Picture name";picture$
11030 LOAD picture$
11040 CLS
11049 REM set window to whole screen and
 print menu again
11050 WINDOW 1,20,1,25
11060 GOSUB 1000
11070 RETURN
12000 WINDOW 1,20,24,25:PEN 1
12010 PRINT "To save a picture"
12020 INPUT "Picture name";picture$
12029 REM save picture from top left loc
ation to bottom right
12030 SAVE picture$,B,&C000,&3FCF
12040 CLS
12050 WINDOW 1,20,1,25
```

```
12060 GOSUB 1000
12070 RETURN
```

Subroutine 12000 carries out the saving of a screen file. It is vital that the picture is not overwritten by messages as the screen is saved, so a window is set up at the bottom of the screen, temporarily obliterating the menu. Line 12030 saves the picture: B indicates a binary file (the format required for saving a section of memory), &C000 is the hexadecimal address of the start of screen memory, which happens to be &3FCF bytes long. Once the file has been saved the menu is redrawn and the program continues.

Subroutine 11000 loads a picture, and is very similar to the save routine, although the LOAD command at 11030 has a much simpler format. Loading a picture is quite intriguing: the picture is not built up from top to bottom as one might imagine, but as a series of widely separated 'strips' across the screen. This is a reflection of the complexity of the screen organisation, where numerically consecutive memory locations are often several screen lines apart.

Exercises

1) Add a 'zoom' option to the menu, so that the picture can be redrawn to a different scale. (You may prefer not to colour-fill any scaled figure, to speed things up.)
2) Add a routine which allows you to input text from the keyboard, and then position it on-screen.
3) Modify the program so that any of the non-flashing colours in mode 0 are available for drawing.

Transformations

Transforming a shape

In earlier chapters we have seen how points can be moved and then fixed once they are in the desired position on-screen. However, we may well wish to move not a single point, but a complete figure, and the movement may not always involve simply shifting the entire figure one pixel in the required direction. In many situations it is useful to be able to carry out a series of TRANSFORMATIONS on a figure.

We have already met the simplest transformation — TRANSLATION, which is the movement of a point or number of points in a single direction. Essentially all our keyboard controls to move up, down, left or right cause the translation of the point or character concerned one pixel in that direction. We can easily extend the previous program so that entire figures can be transformed:

```
1 REM based on the 'drawing progam' of c
hapter 4
2 REM change/add the following lines
3 REM to the earlier program
4 REM note all other options will work b
ut you can't
5 REM change the colour of the figure
6 REM add that for yourself!
63 REM read data for figure
64 REM figure is flag - set to 1 when fi
gure is moved
65 figure=0:moveflag=0:GOSUB 9000
2009 REM only delete point if we're not
moving a figure
2010 IF figure=0 THEN GOSUB 1000
2065 REM fix figure by pressing Space Ba
r
```

```
2070 IF response$=" " THEN IF figure=0 T
HEN GOSUB 3000:linedraw=foregroundcolour
 ELSE figure=0
2077 REM draw/delete figure if flag set
2080 IF figure=1 THEN GOSUB 10000 ELSE G
OSUB 1000
2083 REM draw figure when 'f' pressed
2085 IF response$="f" THEN GOSUB 1000:IF
 moveflag=0 THEN GOSUB 11000:moveflag=1
2086 IF response$="f" THEN x=x(nooflines
):oldx=x:y=y(nooflines):oldy=y:figure=1
8999 REM read data for figure
9000 READ nooflines
9010 FOR count1=1 TO nooflines
9020 READ x(count1),y(count1),l(count1)
9030 NEXT
9040 RETURN
9049 REM data to draw hexagon
9050 DATA 7,300,100,0,400,100,1,490,190,
1,400,280,1,300,280,1,210,190,1,300,100,
1
9997 REM only draw/delete figure from ol
d position
9998 REM if it has been moved
9999 REM translation routine
10000 IF oldx=x AND oldy=y THEN RETURN
10009 REM delete old figure
10010 GOSUB 11000
10019 REM calculate change in position
10020 changex=x-oldx:changey=y-oldy
10030 oldx=x:oldy=y
10039 REM update all coordinates
10040 FOR loop=1 TO nooflines
10050 IF changex=0 THEN y(loop)=y(loop)+
changey ELSE x(loop)=x(loop)+changex
10060 NEXT
10069 REM draw figure to new position
10070 GOSUB 11000
10080 RETURN
10999 REM draw figure routine
11000 FOR loop=2 TO nooflines
11010 IF l(loop)>0 THEN MOVE x(loop),y(l
oop):DRAW x(loop-1),y(loop-1),l(loop)
11020 NEXT
11030 RETURN
```

Translation is the simplest transformation to program, but it is a 'one-off' in the sense that the approach used is not applicable to any other transformations such as the rotation of a figure. A more generalised method is to use MATRICES.

Matrix transformations

We can rotate, enlarge or reflect any figure that we can draw simply by multiplying the coordinates of all the points on that figure by an appropriate matrix. I will give a brief résumé of matrix multiplication, although an understanding of it is not essential to use the programs that follow.

Two matrices can be multiplied by multiplying the numbers in every row of the first matrix by the numbers in every column of the second matrix, as in Figure 7.1. The answer on the right is achieved by adding together the results of the multiplication of each element in the row of the first matrix by the elements in the columns of the second. The first matrix might contain more than one row or more than two columns, but in this case we are considering the transformation of a single point with just two coordinates.

$$\begin{pmatrix}3 & 5\end{pmatrix} \begin{pmatrix}2 & 4 \\ 1 & 2\end{pmatrix} = \begin{pmatrix}3\times2 + 5\times1 & 3\times4 + 5\times2\end{pmatrix}$$

$$= \begin{pmatrix}11 & 22\end{pmatrix}$$

Figure 7.1

A variety of different matrices of the second form can be used to, for example, rotate a point clockwise by 10 degrees, or enlarge a figure by a factor of 1.5, etc. Because the result of the matrix multiplication has the same form as the original 2-element matrix, we can transform the new figure still further if we desire, by multiplying it by a different transformation matrix as in Figure 7.2.

$$\begin{pmatrix}11 & 22\end{pmatrix} \begin{pmatrix}1 & 2 \\ 3 & 2\end{pmatrix} = \begin{pmatrix}11\times1 + 22\times3 & 11\times2 + 22\times2\end{pmatrix}$$

$$= \begin{pmatrix}77 & 66\end{pmatrix}$$

Figure 7.2

$$\begin{pmatrix}2 & 4\\1 & 2\end{pmatrix} \quad \begin{pmatrix}1 & 2\\3 & 2\end{pmatrix} = \begin{pmatrix}2\times1 + 4\times3 & 2\times2 + 4\times2\\1\times1 + 2\times3 & 1\times2 + 2\times2\end{pmatrix}$$

$$= \begin{pmatrix}14 & 12\\7 & 6\end{pmatrix}$$

$$(3\ \ 5) \quad \begin{pmatrix}14 & 10\\7 & 6\end{pmatrix} = (3\times14 + 5\times7 \quad 3\times12 + 5\times6)$$

$$= (77\ \ 66)$$

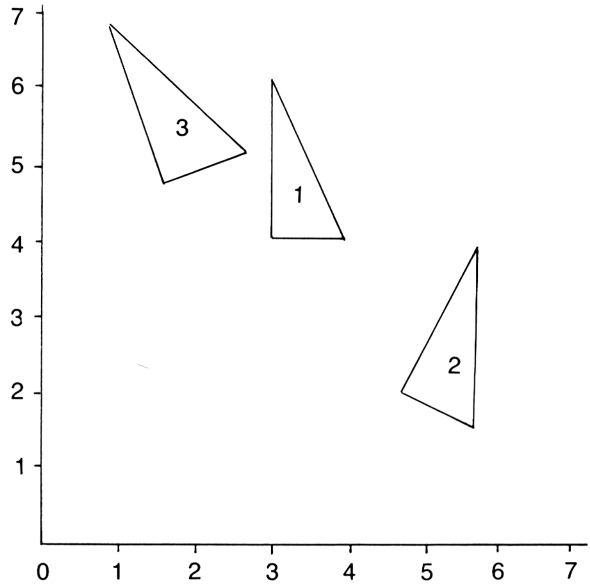
Figure 7.3

In fact we can simplify the process still further. We could get the same result by first multiplying our two transformation matrices together, and using this new matrix to carry out the transformation of the original point all in one go, as in Figure 7.3.

The main disadvantage of using 2×2 transformation matrices is that it is not possible to represent translation as a 2×2 matrix. This prevents us from adopting a completely general approach. Whereas we could reduce the whole series of matrices needed to enlarge, rotate and reflect a figure to a single matrix, translation stands outside the system. We can extend the matrices to 3×3, in which case translation can be incorporated using an appropriate matrix. As we are already familiar with one approach to translation, we shall retain the 2×2 matrices at the cost of having to treat translation as a special transformation not amenable to the same matrix manipulation as the other transformations.

Rotation

Two matrices can be used to rotate a point respectively either clockwise or anticlockwise about the origin, as in Figure 7.4. Comparing the results of the multiplication we can see that the only difference is in the signs of the matrix products. If we denote clockwise rotation by setting the variable 'rotate' to −1, and anticlockwise rotation by setting 'rotate' to 1, we have a single equation that produces the coordinates for rotation in either direction, as in Figure 7.5.



Triangle 1 is rotated clockwise to 2 by 30 degrees by:

$$\begin{pmatrix} 3 & 4 \\ 4 & 4 \\ 3 & 6 \end{pmatrix} \quad \begin{pmatrix} \cos 30 & -\sin 30 \\ \sin 30 & \cos 30 \end{pmatrix} \quad = \begin{pmatrix} 4.6 & 2.0 \\ 5.5 & 1.5 \\ 5.6 & 3.7 \end{pmatrix}$$

Triangle 1 is rotated anticlockwise to 3 by 20 degrees by:

$$\begin{pmatrix} 3 & 4 \\ 4 & 4 \\ 3 & 6 \end{pmatrix} \quad \begin{pmatrix} \cos 20 & \sin 20 \\ -\sin 20 & \cos 20 \end{pmatrix} \quad = \begin{pmatrix} 1.5 & 4.8 \\ 2.4 & 5.1 \\ 0.8 & 6.7 \end{pmatrix}$$

In a more general form rotation clockwise by θ degrees is given by:

$$(x\ y) \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} = (x\cos\theta + y\sin\theta - x\sin\theta + y\cos\theta)$$

and rotation anticlockwise by degrees:

$$(x\ y) \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = (x\cos\theta + y\sin\theta - x\sin\theta + y\cos\theta)$$

Figure 7.4

$$newx = oldx \times \cos\theta + rotate \times oldy \times \sin\theta$$
$$newy = -rotate \times oldx \times \sin\theta + oldy \times \sin\theta$$

Figure 7.5

We can extend the previous program that enabled us to translate a figure so that it can also be rotated:

```
2083 REM draw figure when 'f' pressed
2084 REM turn figure when 't' pressed
2085 IF response$="f" OR response$="t" T
HEN GOSUB 1000:IF moveflag=0 THEN GOSUB
11000:moveflag=1
2086 IF response$="f" OR response$="t" T
HEN x=x(nooflines):oldx=x:y=y(nooflines)
:oldy=y:IF response$="f" THEN figure=1
2087 IF response$="t" THEN GOSUB 12000
12000 REM we'll use this in a minute!
12007 REM set COS and SIN to accept degr
ees
12008 REM rotation is at 5 degree interv
als
12009 REM for a different interval chang
e lines 12020,30,100
12010 DEG
12020 transformx=COS(5)
12030 transformy1=COS(5)
12040 rotstop$=""
12049 REM continue turning figure until
't' pressed again
12050 WHILE rotstop$<>"t"
12060 rotstop$=LOWER$(INKEY$)
12069 REM set 'rotate' to indicate direc
tion of turn
12070 rotate=0
12079 REM turn anti- or clockwise if 'a'
 or 'c' pressed
12080 IF rotstop$="a" THEN rotate=-1
12090 IF rotstop$="c" THEN rotate=1
12100 IF rotate<>0 THEN GOSUB 11000:tran
sformy=SIN(5)*rotate:transformx1=-SIN(5)
*rotate:GOSUB 13000:GOSUB 11000
12110 WEND
12119 REM restore original point
12120 GOSUB 1000
12130 RETURN
12998 REM general transformation routine
12999 REM (except for the special case o
f translation)
13000 FOR loop=1 TO nooflines
13010 x1=x(loop)-centrex:y1=y(loop)-cent
rey
```

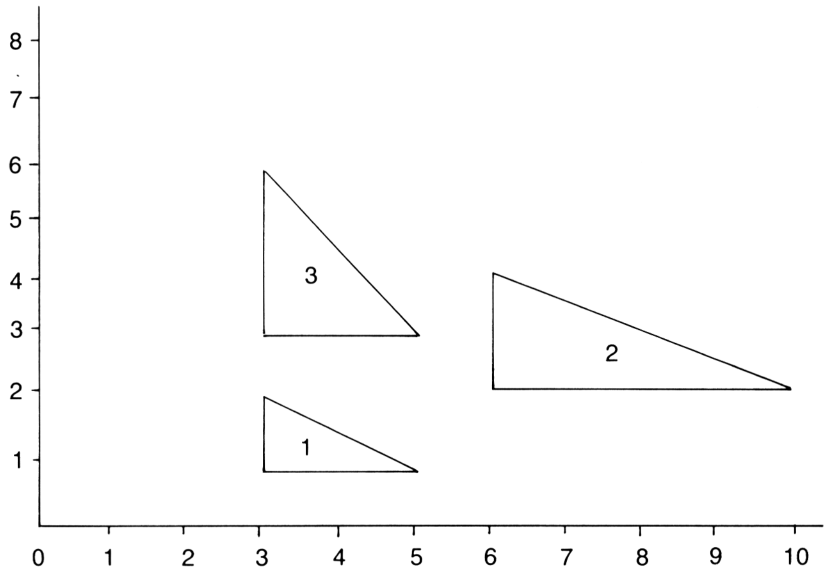
```
13020 x(loop)=transformx*x1+transformy*y
1+centrex
13030 y(loop)=transformx1*x1+transformy1
*y1+centrey
13040 NEXT
13050 RETURN
```

Note that rotation is about (0,0) only. It is more useful to be able to choose our own centre of rotation, and this is fairly easy to cater for:

```
12000 GOSUB 14000
13997 REM to set centre for rotation
13998 REM notice similarity to 2030-60
13999 REM could be rewritten as another
subroutine
14000 centre$=""
14010 GOSUB 1000
14019 REM scan keyboard until centre fix
ed by 'f'
14020 WHILE centre$<>"f"
14030 GOSUB 1000
14040 centre$=LOWER$(INKEY$)
14049 REM move centre up/down/left/right
/etc
14050 IF centre$="a" THEN y=y+2
14060 IF centre$="z" THEN y=y-2
14070 IF centre$="," THEN x=x-4
14080 IF centre$="." THEN x=x+4
14090 GOSUB 1000
14100 WEND
14109 REM record centre coordinates and
delete point
14110 centrex=x:centrey=y:GOSUB 1000
14120 RETURN
```

Enlargement and reduction

We saw in an earlier chapter that enlargement or reduction of a figure was relatively straightforward, although at the time we had no control over the centre of enlargement. Using the matrix method we can introduce some interesting enlargements involving a variation in the scale factor in the x and y directions, which will stretch the figure concerned along its x or y axis, as in Figure 7.6. Let's add this facility to our program:



Triangle 1 can be enlarged to 2 by:

$$\begin{pmatrix}3 & 1\\5 & 1\\3 & 2\end{pmatrix} \quad \begin{pmatrix}2 & 0\\0 & 2\end{pmatrix} \quad = \begin{pmatrix}6 & 2\\10 & 2\\6 & 4\end{pmatrix}$$

Differing values on the diagonal result in 'stretches' parallel to one of the axes:

$$\begin{pmatrix}3 & 1\\5 & 1\\3 & 2\end{pmatrix} \quad \begin{pmatrix}1 & 0\\0 & 3\end{pmatrix} \quad = \begin{pmatrix}3 & 3\\5 & 3\\3 & 6\end{pmatrix}$$

In a more general form, scaling or stretching is given by:

$$(x\ y) \begin{pmatrix}n1 & 0\\0 & n2\end{pmatrix} = (n1 \times x \quad n2 \times y)$$

Figure 7.6

```
2077 REM draw/delete figure if flag set
2080 IF figure=1 THEN GOSUB 10000 ELSE G
OSUB 1000
2082 REM scale figure when 's' pressed
2083 REM draw figure when 'f' pressed
2084 REM turn figure when 't' pressed
2085 IF response$="f" OR response$="t" O
R response$="s" THEN GOSUB 1000:IF movef
lag=0 THEN GOSUB 11000:moveflag=1
2086 IF response$="f" OR response$="t" O
R response$="s" THEN x=x(nooflines):oldx
```

```
=x:y=y(nooflines):oldy=y:IF response$="f
" THEN figure=1
2087 IF response$="t" THEN GOSUB 12000
2088 IF response$="s" THEN GOSUB 15000
15000 REM we'll use this in a minute
15010 transformy=0
15020 transformx1=0
15030 scalestop$=""
15039 REM continue scaling figure until
's' pressed
15040 WHILE scalestop$<>"s"
15050 scalestop$=LOWER$(INKEY$)
15057 REM scale factors can be changed b
y
15058 REM modifying the values in lines
15070,80
15059 REM transformx set to indicate sca
ling
15060 transformx=0
15069 REM enlargement by 1.1 if 'e' pres
sed
15070 IF scalestop$="e" THEN transformx=
1.1:transformy1=1.1
15079 REM reduction by 0.9 if 'r' presse
d
15080 IF scalestop$="r" THEN transformx=
0.9:transformy1=0.9
15090 IF transformx<>0 THEN GOSUB 11000:
GOSUB 13000:GOSUB 11000
15100 WEND
15110 GOSUB 1000
15120 RETURN
```

As before, it is better if we can select the centre of enlargement/reduction:

```
15000 GOSUB 14000
```

Reflection

Reflection in the x or y axis can be carried out using the matrices shown in Figure 7.7. As with rotation, these are essentially the same matrix. We can only see the reflection if we move the axes, of course, otherwise the result will be drawn off-screen!

$$(x\ y)\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} = (x\ -y) \qquad \text{(reflection in the x axis)}$$

$$(x\ y)\begin{pmatrix}-1 & 0\\ 0 & 1\end{pmatrix} = (-x\ y) \qquad \text{(reflection in the y axis)}$$

Figure 7.7

```
2081 REM mirror image when 'm' pressed
2082 REM scale figure when 's' pressed
2083 REM draw figure when 'f' pressed
2084 REM turn figure when 't' pressed
2085 IF response$="f" OR response$="t" O
R response$="s" OR response$="m" THEN GO
SUB 1000:IF moveflag=0 THEN GOSUB 11000:
moveflag=1
2086 IF response$="f" OR response$="t" O
R response$="s" OR response$="m" THEN x=
x(nooflines):oldx=x:y=y(nooflines):oldy=
y:IF response$="f" THEN figure=1
2087 IF response$="t" THEN GOSUB 12000
2088 IF response$="s" THEN GOSUB 15000
15996 REM the axis of reflection can onl
y be horizontal
15997 REM or vertical - indicate its pos
ition by
15998 REM moving the point and then pres
s 'x' or 'y'
15999 REM to select the axis
16000 GOSUB 14000
16010 axis$=""
16020 WHILE axis$<>"x" AND axis$<>"y"
16030 axis$=LOWER$(INKEY$)
16040 IF axis$="x" THEN transformx=1:tra
nsformy1=-1
16050 IF axis$="y" THEN transformx=-1:tr
ansformy1=1
16060 WEND
16070 GOSUB 11000:GOSUB 13000:GOSUB 1100
0
16080 GOSUB 1000
16090 RETURN
```

Reflection in a specified line is trickier, and involves other transformations.

Shearing

Shearing involves a movement parallel to the x or y axis: the amount of shear at a point depends upon the distance from the shear axis:

```
2084 REM 'push' figure (shear) when 'p'
pressed
2085 IF response$="f" OR response$="t" O
R response$="s" OR response$="m" OR resp
onse$="p" THEN GOSUB 1000:IF moveflag=0
THEN GOSUB 11000:moveflag=1
2086 IF response$="f" OR response$="t" O
R response$="s" OR response$="m" OR resp
onse$="p" THEN x=x(nooflines):oldx=x:y=y
(nooflines):oldy=y:IF response$="f" THEN
 figure=1
2087 IF response$="t" THEN GOSUB 12000
2088 IF response$="s" THEN GOSUB 15000
2089 IF response$="m" THEN GOSUB 16000
2090 IF response$="p" THEN GOSUB 17000
2095 WEND
16998 REM shear routine very similar to
reflection
16999 REM again these could be combined
to one routine
17000 GOSUB 14000
17010 axis$=""
17017 REM shear by 1 unit for every unit
 distance
17018 REM from the axis - you may find t
his too much
17019 REM if so make 17020,30 fractional
17020 transformx=1
17030 transformy1=1
17039 REM scan keyboard until x or y axi
s selected
17040 WHILE axis$<>"x" AND axis$<>"y"
17050 axis$=LOWER$(INKEY$)
17060 IF axis$="x" THEN transformx1=0:tr
ansformy=1
17070 IF axis$="y" THEN transformx1=1:tr
ansformy=0
17080 WEND
```

Exercises

1) The transformations in the program are all carried out on a figure whose coordinates are read from DATA statements. Add routines to the transformation program so that you can first draw a figure on-screen and then transform it.
2) Modify the program to include an option so that the previous positions of the figure remain on-screen when it is transformed. (This can produce some interesting patterns.)
3) The various transformation matrices can also be used to good effect to produce patterns. We touched on this in Chapter 4, when we saw that repetitive rotation and enlargement of a figure gives some striking results. Write a program which allows you to specify a transformation or series of transformations to be carried out on a figure. You are also able to choose the number of times the transformation will be repeated on the new figure which is drawn. Once you have completed your specification, the computer carries out the transformations repeatedly, and displays all the figures which result. You might like to draw each figure in a different colour for a better effect.
4) CAD (Computer-Aided Design) programs often contain standard shapes on a menu. The user can 'pick up' the shape from the menu, when it becomes attached to the cursor, move it to a position, and then fix the shape. Write a simple CAD program which simplifies the drawing of a house. Include as standard shapes several types of door and window. (Add enough detail so that each of these are different, but not so much that it takes an inordinately long time to be redrawn at a new position.)

Index

Other titles available from Micro Press:

15 GRAPHIC GAMES FOR THE SPECTRUM

Richard G. Hurley
0 7447 0002 7

GRAPHIC ADVENTURES FOR THE SPECTRUM 48K

Richard G. Hurley
0 7447 0013 2

SPECTRUM SUPERGAMES

Richard G. Hurley
0 7447 0017 5

MAKING THE MOST OF YOUR SPECTRUM MICRO DRIVES

Richard G. Hurley
0 7447 0005 1

THE SPECTRUM OPERATING SYSTEM

Steve Kramer
0 7447 0019 1

MASTERING THE TI-99

Peter Brooks
0 7447 0008 6

ADVANCING WITH THE ELECTRON

Peter Seal
0 7447 0012 4

QUALITY PROGRAMS FOR THE ELECTRON

Simon
0 7447 0004 3

THE ATMOS BOOK OF GAMES

Wynford James
0 7447 0018 3

QL SUPERBASIC: A PROGRAMMER'S GUIDE

John Wilson
0 7447 0020 5

THE QL BOOK OF GAMES

Richard G. Hurley
0 7447 0022 1

QUALITY PROGRAMS FOR THE BBC MICRO

Simon
0 7447 0001 9

EDUCATIONAL GAMES FOR THE BBC MICRO

Ian Soutar
0 7447 0016 7

INTERFACING AND ROBOTICS ON THE BBC MICRO

Ray Bradley
0 7447 0023 X

BBC MICRO DISK DRIVES

R. D. Bagnall
0 7447 0028 0

BASIC PROGRAMMING ON THE AMSTRAD

Wynford James
0 7447 0024 8

MACHINE CODE FOR BEGINNERS ON THE AMSTRAD

Steve Kramer
0 7447 0025 6

THE COMMODORE 64 BOOK OF SOUND AND GRAPHICS

Simon
0 7447 0015 9

BASIC PROGRAMMING ON THE COMMODORE 64

Gordon Davis & Fin Fahey
0 7447 0026 4

PLUS/4 MAGIC FOR BEGINNERS

Bill Bennett
0 7447 0031 0

GRAPHICS PROGRAMMING TECHNIQUES ON THE AMSTRAD CPC 464

Good graphics are central to many programs and this book describes how you can exploit the excellent facilities offered by the Amstrad CPC 464 to the full. The example programs include routines which you can readily incorporate into your own software.

Areas covered include arcade games and the animation of simple figures, drawing and saving of colourful pictures, the construction of bar charts and pie charts, the scaling and transformation of shapes, and many other exciting applications. Every chapter includes suggestions for further programming based on the examples provided.

The Author

Wynford James writes education material (including software) for a major microcomputer company. Prior to that he was a technical author for ICL. He has also taught mathematics and was actively involved in the development of computer studies throughout his school.

GB £ NET +007.95

ISBN 0-7447-0027-2

00795

9 780744 700275

GRAPHICS PROGRAMMING TECHNIQUES ON THE AMSTRAD CPC 464

James

MICRO PRESS