CHAPTER SEVEN

GENERAL GRAPHICS

The graphics system on the BBC Micro is based on two chips. These are the 6845 Cathode Ray Tube Controller (CRTC for short) and the Video ULA. Between them, these two chips are responsible for the flexibility that allows the BBC Micro to have eight different screen modes.

To understand how to use these chips we must first look at the way in which a TV monitor works. Inside a TV is a tube with the screen at one end of it. In this tube a narrow beam of electrons is fired at the screen. The screen is covered on the inside with a substance which glows where the beam hits it. The beam can obviously only illuminate one dot on the screen at once and yet we need large amounts of screen to be lit, seemingly continuously. What happens is that the beam is scanned across the screen in a series of horizontal lines from left to right starting at the top and working down. At the same time the beam is switched on and off to produce light and dark areas on the screen. The beam completes one vertical scan of the screen every fiftieth of a second, so the eye is fooled into seeing a coherent picture.

The graphics registers

The 6845 CRTC contains 18 registers that we can use. To write to them, the number of the register we want to access (from 0 to 17) must be placed at address &FE00 and the data can then be written through location &FE01. The legal way to write to a register is through the operating system, using VDU23. The format is like this:

There is no way to read from any register. There is a complete description of all these registers in THE ADVANCED USER GUIDE. For the purposes of this book we will only look at some of the more useful registers.

Register 1 gives the total number of displayed characters per line.

However, although we would assume that in Mode 2, say, it would contain 20, this is not so. The 6845 CRTC was not designed to be used for high resolution graphics − it was originally designed as a straightforward text VDU controller, for text like that of Mode 7, with only one byte needed to store each character, though it is very simple to use it for black-and-white graphics where each character takes up eight bytes. So to use it as the heart of a colour display as required by the BBC Micro, it must be made to address and fetch the right number of bytes of data for each text character in that mode. For Mode 2 each text character takes up all of 32 bytes − not the eight bytes that a normal black-and-white display would need to store a character. The result is that for each text character on the Mode 2 colour screen the 6845 needs to fetch four of its eight-byte characters. This means that, in the 20-character Mode 2, the CRTC is made to think that there are 80 of ITS characters per line. Thus, for Modes 0 to 3, Register 1 contains 80, and for Modes 4 to 7 it contains 40.

By changing Register 1 you can make narrower screens. For instance, suppose you have a game that needs Mode 4 but is very long. If you are prepared to only use 256 pixels across the screen then you can set Register 1 to 32. This will mean that the screen only takes up 8K instead of 10K. This technique is used by Acornsoft to fit ELITE into the computer.

Register 2 contains the position of the horizontal sync pulse.

This is effectively used to position the screen on

your monitor, left and right. In different modes it contains these values:

Mode	0	1	2	3	4	5	6	7
Register 2	98	98	98	98	49	49	49	51

If you are using the narrow 8K Mode 4 you will need to set this register to 45 instead of 49.

Register 6 is the vertical equivalent of Register 1.

It gives the number of vertical text lines to a screen. Its normal values for the modes are:

Mode	0	1	2	3	4	5	6	7
Register 6	32	32	32	25	32	32	25	25

This can be used together with Register 1 to produce smaller modes.

Register 7 contains the vertical sync position.

It is used by the *TV command to move the screen up and down to accommodate different monitors. As before, for smaller modes you may have to change this register.

Registers 12 and 13 give the start of screen address in RAM.

Notice that Register 12 is high and Register 13 is low! These two registers form the address of the top-left hand corner of the sceen in RAM, but DIVIDED BY EIGHT. For example, in Mode 2, where the top left-hand corner is stored at &3000, these registers contain &600. By changing these registers you can make any part of the RAM the screen. Also, the registers can be used for scrolling. If the registers are set so that there is not enough addressed RAM after the top left-hand corner to display a complete screen − that is, the address of the bottom of the screen will occur after &7FFF − then the hardware wraps the screen back to before the top left-hand corner.

For example, if in Mode 2 we set Registers 12 and 13 to &B00, i.e. pointing to &5800 − exactly half way down the screen − then, when the top half of the

screen has been displayed the VDU will go back to &3000 to display the bottom half. This is how scrolling is done in all modes, as this saves moving large amounts of memory around. The point in the memory to which the screen scrolls back is normally dependent on the mode and is set to the normal top-of-screen address. However, this is set as part of changing mode through the system VIA.

Bits 0 to 3 of port B on the system VIA are used as an addressable latch to control several functions of the BBC Micro. We are interested in controlling the scroll wrap-around. This control is achieved by altering the settings of two bits. The explanation of these is a bit (sic) complicated, so I shall just give the four relevant commands:

?&FE40=&4	clears low bit
?&FE40=&C	sets low bit
?&FE40=&5	clears high bit
?&FE40=&D	sets high bit

These two bits in their four combinations set the start of screen RAM for scrolling, as follows:

high	low	address
0	0	&4000
0	1	&6000
1	0	&5800
1	1	&3000

So we now have all we need to set up an 8K 'narrow' Mode 4.

```
10 MODE4
20 VDU23,0,1,32;0;0;0
30 VDU23,0,2,45;0;0;0
40 VDU23,0,12,12;0;0;0
50 ?&FE40=&5:?&FE40=&C
60 ?&34E=&60:?&351=&60
70 VDU30
80 HIMEM=&6000
```

This first adjusts Register 1 to set the number of characters per line to 32 (line 20). Line 30 then

adjusts Register 2 to shift the whole screen right four characters to centre it. Next the top-of-screen is set to &6000 (Line 40) and the scrolling is set to wrap around to &6000 (line 50). Finally, printing to the screen is set to start at &6000 (line 60) by setting the operating system's top-of-screen variables suitably and homing the cursor (line 70). We can now claim the extra 2K of memory by moving HIMEM up to &6000 (line 80).

However, this technique leaves the operating system somewhat confused. It continues trying to print the normal Mode 4 40-column text and 320-pixel-wide graphics. To use the system you will have to use the PRINT and PLOT commands very carefully!

The Video ULA

The Video ULA is accessed through two write-only locations. The Video Control Register at &FE20 controls a number of factors to do with the current mode. Complete details are given in THE ADVANCED USER GUIDE. As far as we are concerned it just sets which mode we are in.

Much more difficult to explain is the second register at &FE21 which gives us access to the palette. The high nibble of the location is the logical colour and the low nibble is the actual colour. By sending one byte to this location we can change one entry in the palette. So far, so good, but here comes the crunch. Just to make things more difficult, there are two complications. The first is that the actual colour nibble should contain the actual colour Exclusive-ORed with 7! The second is that to change all the colours in ANY mode you have to alter 16 entries in the palette.

For the 16-colour Mode 2 this is obvious. Each logical colour has only one entry in the palette.

For the one-colour modes, entries 0 to 7 in the palette all be set according to logical colour 0 and entries 8 to 15 must be set according to logical colour 1.

For four-colour modes, it is worse. To set each logical colour you need to set four entries as follows:

Logical colour	Entries to be changed
0	0
	1
	4
	5
1	2
	3
	6
	7
2	8
	9
	12
	13
3	10
	11
	14
	15

For example, in Mode 1, to do the equivalent of VDU19,2,3;0; we should have to do:

```
?&FE21=&84
?&FE21=&94
?&FE21=&C4
?&FE21=&D4
```

The reasons for all this are so involved that they are virtually unexplainable. However, I can assure you that there are very good reasons − I think! As far as we are concerned it is usually easier to use VDU19. However, we will see, later in this chapter, one example of where the added speed of the direct method is necessary. Notice that this method won't alter the copy of the palette that the operating system keeps, so that using OSWORD 11 to read the palette will give false answers.

Screen splitting

It would be very useful to be able to split the screen into two halves and have one half, say, in Mode 0 and the other half in Mode 1. This may seem impossible; but, using interrupts, it can be done!

What we need to do is to use the vertical sync interrupt to put the screen in Mode 0 and then set a timer to produce an interrupt, half-way down the screen, which we can use to put the screen in Mode 1. Because both are 20K modes it shouldn't be too difficult to change mode in the middle. The contents of the 6845 shouldn't need any changes, so all we need to change is the contents of the ULA.

Let's write two routines that will put the screen in Mode 1 from Mode 0 and Mode 0 from Mode 1 without clearing the screen. First of all, let us assume that the screen is already in Mode 0. To change to Mode 1, the first thing we need to do is change the video control register at &FE20 to set up the mode we are using. We also need to update the operating system's copy of this register at &248

```
.mode1   LDA #&D8
         STA &FE20
         STA &248
```

(320-340)

Next we need to change the palette by writing to the register at &FE21. Colours 0 and 3 will already be correctly set so we only need to change 1 and 2. We need to change four bytes for each logical colour in Mode 1.

```
         LDX #7
.l1      LDA m1,X
         STA &FE21
         DEX
         BPL l1
         RTS
.m1      EQUD &26366676
         EQUD &8494C4D4
```

(360-420)

We can do likewise for Mode 0:

```
.mode0   LDA #&9C
         STA &FE20
```

```
        STA &248
        LDX #7
.l0     LDA m0,X
        STA &FE21
        DEX
        BPL l0
        RTS
.m1     EQUD &27376777
        EQUD &8090C0D0
```

(200-300)

Next we need to set up an interrupt routine in the interrupt vector (&204 and &205).

```
.init   SEI
        LDA &204
        STA &230
        LDA &205
        STA &231
        LDA #irq MOD256
        STA &204
        LDA #irq DIV256
        STA &205
```

(440-520)

We will also need to disable the centisecond clock interrupt and the analogue-to-digital converter interrupt, as these will tend to interfere with the program. If you need to leave these enabled, then you will have to put up with some flickering on the screen.

```
        LDA #&50
        STA &FE4E
        CLI
        RTS
```

(530-560)

The interrupt routine will first need to save the registers.

```
.irq    LDA &FC
        PHA
        TXA
        PHA
        TYA
        PHA

(580-630)
```

Next we need to check that the vertical sync caused the interrupt. If so, we need to switch to Mode 0.

```
        LDA #2
        BIT &FE4D
        BEQ notsync
        JSR mode0

(640-670)
```

We then need to set Timer 2 in the system VIA to count for half a screen and then produce an interrupt. So that we can move the boundary between the two modes easily, we put the boundary position in &70 and &71.

```
        LDA &FE4B
        AND #&DF
        STA &FE4B
        LDA &FE4E
        ORA #&20
        STA &FE4E
        LDA &70
        STA &FE48
        LDA &71
        STA &FE49

(680-770)
```

Finally we need to exit the interrupt routine by restoring the registers and jumping to the spare vector.

```
.exit   PLA
        TAY
```

```
        PLA
        TAX
        PLA
        STA &FC
        JMP (&230)

(780-840)
```

If the interrupt is not caused by the vertical sync then we need to check whether it is the timer interrupt. If so, we must clear the interrupt flag and switch to Mode 1.

```
.notsync LDA #&20
        BIT &FE4D
        BEQ exit
        STA &FE4D
        JSR mode1
        JMP exit

(850-900)
```

Now all that remains is to try an example:

```
10 *TV0,1
20 MODE0
30 MOVE0,512:DRAW1279,512
```

Notice that, because the change of modes itself takes some time, about one line of pixels on the screen will be garbled. To overcome this we must make this line white so that it is the same in both modes. This way, the transition should be invisible.

```
40 PROCass
50 !&70=9900
```

This sets up the position of the border. It may vary from machine to machine, so you may have to experiment to find the best value.

```
60 CALLinit
```

Finally we can draw a pattern on the screen as a

demonstration.

```
 70 FORA%=0TO1:VDU29,A%*640;512;
 80 FORB%=0TO511STEP16
 90 MOVEB%*1.25,0:DRAW639,B%
100 DRAW639-B%*1.25,511:DRAW0,511-B%
110 DRAWB%*1.25,0:NEXT,
120 VDU26:FORA%=0TO1279STEP2
130 IFRND(2)=2MOVEA%,0:DRAWA%,508
140 NEXT
150 GOTO150
160 DEFPROCass
170 FORpass%=0TO2STEP2
180 P%=&A00
190 [OPTpass%
200 .mode0   LDA #&9C        \ Change to
210          STA &FE20       \ Mode 0.
220          STA &248
230          LDX #7
240 .l0      LDA m0,X
250          STA &FE21
260          DEX
270          BPL l0
280          RTS
290 .m0      EQUD &8090C0D0  \ Colour data
300          EQUD &27376777  \ for Mode 0.
310
320 .mode1   LDA #&D8
330          STA &FE20
340          STA &248
350          LDX #7
360 .l1      LDA m1,X
370          STA &FE21
380          DEX
390          BPL l1
400          RTS
410 .m1      EQUD &26366676  \ Colour data
420          EQUD &8494C4D4  \ for Mode 1.
430
440 .init    SEI             \ Initialise
450          LDA &204        \ interrupts
460          STA &230
470          LDA &205
480          STA &231
```

```
490          LDA #irq MOD256
500          STA &204
510          LDA #irq DIV256
520          STA &205
530          LDA #&50
540          STA &FE4E
550          CLI
560          RTS
570
580 .irq     LDA &FC         \ Trap ints.
590          PHA
600          TXA
610          PHA
620          TYA
630          PHA
640          LDA #2          \ Check for
650          BIT &FE4D       \ v.sync.
660          BEQ notsync
670          JSR mode0       \ Change to
680          LDA &FE4B       \ Mode 0
690          AND #&DF        \ and start
700          STA &FE4B       \ counter for
710          LDA &FE4E       \ boundary.
720          ORA #&20
730          STA &FE4E
740          LDA &70
750          STA &FE4B
760          LDA &71
770          STA &FE49
780 .exit    PLA
790          TAY
800          PLA
810          TAX
820          PLA
830          STA &FC
840          JMP (&230)
850 .notsync LDA #&20         \ Check for
860          BIT &FE4D        \ Timer.
870          BEQ exit
880          STA &FE4D
890          JSR mode1        \ Change to
900          JMP exit         \ Mode 1.
910 ]
920 NEXT
```

Beware of using VDU19. You will also find that plotting lines in the bottom half of the screen has interesting effects.

Screen swapping

Some expensive computers nowadays have a system which allows for totally flicker-free animation. This system has two completely separate blocks of memory for graphics. While one image is being displayed the other, concealed, image is being redrawn. When the new image is complete the VDU switches cleanly between the two pages so that the new image is displayed, while the first is redrawn. By repeating this process the image need never be seen being redrawn.

By now you will have guessed that there is a method for doing this on the BBC Micro. Because two complete blocks of memory are needed, we can't use a 20K mode. For our purposes let's try and animate two Mode 4 screens. The two blocks of memory, each 10K long, will start at &3000 and &5800 respectively.

The first problem we need to look at is that before we can redraw on the concealed screen we must clear its 10K block. All we have to do is set 10K of memory to zero. This sounds like a simple task using post-indexed indirect addressing. However, for clearing such a large amount of memory this addressing mode is too slow. We will need to sacrifice the short, neat but slow solution for the long but fast solution. By using absolute indexed addressing we can clear one 256-byte page very fast, like this:

```
.clear   LDA #0
         TAX
.loop    STA &3000,X
         INX
         BNE loop
         RTS
```

To clear 10K of memory we need to add an STA command for each page of memory we wish to clear

− here, this means 40 STA commands. To save typing each of these separately, we can use the power of the assembler. To start with, we need the first few commands. Notice that we disable interrupts first, to increase speed that extra little bit.

```
1000 DEFPROCass
1010 DIMmc%500
1020 FORpass%=0TO2STEP2
1030 P%=mc%
1040 [OPTpass%
1050 .lclear  SEI
1060          LDA #0
1070          TAX
1080 .lloop
1090 ]
```

Next we set A% to a loop from &3000 to &5700 in steps of 256. Then we re-enter the assembler where we left off, reset the option, and set up the STA command with the variable A%. Then we exit the assembler again and end the loop.

```
1100 FORA%=&3000TO&5700STEP256
1110 [OPTpass%
1120          STA A%,X
1130 ]:NEXT
```

The result of this is that we have assembled all 40 commands as required. This has shortened the assembly code considerably and shows the advantages of a powerful assembler! Finally we must end the machine code loop, and re-enable the interrupts.

```
1140 [OPTpass%
1150          INX
1160          BNE lloop
1170          CLI
1180          RTS
```

We can write a similar routine for clearing the second 10K block.

```
1190 .hclear  SEI
1200          LDA #0
1210          TAX
1220 .hloop
1230 ]
1240 FORA%=&5800TO&7F00STEP256
1250 [OPTpass%
1260          STA A%,X
1270 ]:NEXT
1280 [OPTpass%
1290          INX
1300          BNE hloop
1310          CLI
1320          RTS
```

Now we can write a routine which swaps the two screens around. The obvious way to do this is to swap the two sections of memory. However, this would be ridiculously slow. Instead, we will switch the start-of-screen RAM address register in the 6845 between the two blocks, so altering which block is being displayed.

We will need a variable as a flag that will tell us which of the two blocks is currently being displayed. We can use &70 for this − if it contains zero then the low block is currently being displayed; otherwise, it contains 255.

The first job the routine must do is to wait for the vertical sync so that the screens swap cleanly during the vertical sync period.

```
1330 .swap    LDA #19
1340          JSR &FFF4
```

Next we need to invert the contents of &70 by EORing it with 255. If it is now 255 then we want to swap to displaying the high block.

```
1350          LDA &70
1360          EOR #255
1370          STA &70
1380          BNE high
```

We now want to display the low block. To do this

we must alter the start-of-display RAM register to point to the low block (we only need change the high byte as the low byte is zero in both cases).

```
1390          LDA #12
1400          STA &FE00
1410          LDA #8
1420          STA &FE01
```

Next we need to ensure that any plotting or printing will be placed where it can't be seen in the high block. To do this we need to alter two bytes in the operating system work-space to the high byte of the address of the top left-hand corner of the screen on which we are plotting − here, &58. We also need to do a cursor-home to move all the cursors, etcetera. At this point all that remains is to clear the concealed screen ready for plotting.

```
1430          LDA #&58
1440          STA &34E
1450          STA &351
1460          LDA #30
1470          JSR &FFEE
1480          JMP hclear
```

Now we can use the same method to swap back again.

```
1490 .high    LDA #12
1500          STA &FE00
1510          LDA #11
1520          STA &FE01
1530          LDA #&30
1540          STA &34E
1550          STA &351
1560          LDA #30
1570          JSR &FFEE
1580          JMP lclear
1590 ]
1600 NEXT
1610 ENDPROC
```

Now all that we need is in example of how to use the program.

```
10 MODE4:VDU23,1,0;0;0;0;
20 HIMEM=&3000
30 PROCass
40 CALLlclear:?&70=255:CALLswap
```

Notice that we need to set HIMEM to reserve 20K of screen memory, the lower 10K of which we need to clear. Note also that we need to call SWAP once just to get everything running smoothly − this saves having an initialisation routine.

```
50 FORA%=0TO1019STEP16:MOVEA%,0
60 DRAW1023,A%:DRAW1023-A%,1023
70 DRAW0,1023-A%:DRAWA%,0
80 CALLswap:NEXT
90 GOTO50
```

A BASIC swap

Of course, if you don't need to redraw the image completely each time, and hence don't need to clear the screen, you don't necessarily need to use machine code. So, just to show that good results don't always need machine code (though it helps), here is an analogue clock entirely in BASIC.

The program works, as before, in two Mode 4 screens. The first job is to write a procedure to draw a face without the hands.

```
1000 DEFPROCface
1010 GCOL0,1:P%=4
1020 FORA=0TOPI*2STEPPI/24
1030 MOVE0,0:PLOTP%,512*SINA,512*COSA
1040 P%=85:NEXT
1050 GCOL0,0:P%=4
1060 FORA=0TOPI*2STEPPI/24
1070 MOVE0,0:PLOTP%,492*SINA,492*COSA
1080 P%=85:NEXT
```

This draws the rim of the face. Next we need the dots for the hours and a dot at the centre. For speed it is better to use a defined character for this and

position it with the VDU 5 'text at graphics cursor' mode.

```
1090 VDU23,224,0,&1C,&3E,&7F,&7F,&7F,
     &3E,&1C,5
1100 GCOL0,1:FORA=0TOPI*2STEPPI/6
1110 MOVE450*SINA-16,450*C0SA+16:VDU224
1120 NEXT:MOVE-16,16
1130 VDU224,4,23,1,0;0;0;0;
1140 ENDPROC
```

Next we need a procedure that will draw the three hands in their correct positions. Notice that we have not specified the colour in this procedure, so it can be used both to draw the hands and remove them.

2000 DEFPROChands(hours,minutes,seconds)

```
2010 A=seconds*PI/30:X=SINA:Y=COSA
2020 MOVEX*32,Y*32:DRAWX*420,Y*420
2030 A=minutes*PI/30:X=SINA:Y=COSA
2040 MOVEX*32+Y*5,Y*32-X*5
2050 MOVEX*32-Y*5,Y*32+X*5
2060 PLOT85,X*370+Y*5,Y*370-X*5
2070 PLOT85,X*370-Y*5,Y*370+X*5
2080 A=hours*PI/6:X=SINA:Y=COSA
2090 MOVEX*32+Y*12,Y*32-X*12
2100 MOVEX*32-Y*12,Y*32+X*12
2110 PLOT85,X*300+Y*12,Y*300-X*12
2120 PLOT85,X*300-Y*12,Y*300+X*12
2130 ENDPROC
```

Now we need the BASIC equivalent of the swap routine from the machine code program, but without the clear routines. We can use the variable S% instead of &70.

```
3000 DEFPROCswap
3010 *FX19
3020 S%=S%EOR1
3030 IFS%THEN?&FE00=12:?&FE01=6:
     ?&34E=&58:?&351=&58:VDU30:ENDPROC
3040 ?&FE00=12:?&FE01=11:?&34E=&30:
     ?&351=&30:VDU30:ENDPROC
```

Now we can write the main routine. The first job is to input the start time.

```
10 MODE7:INPUT"Hours, Minutes,
   Seconds",hours,minutes,seconds
```

Next we need to clear both screens. As we have no machine-code clearing routines the easiest way to do this is to go into Mode 0. Next we need to go into Mode 4 and set HIMEM to reserve the 20K of screen memory.

```
20 MODE0:MODE4:HIMEM=&3000
```

Now we must turn off the cursor and set the graphics origin to the middle of the screen. We must also set S% suitably.

```
30 S%=0:VDU23,1,0;0;0;0;29,640;512;
```

Next we need to draw the face on each of the two pages, having first called SWAP to set the paged graphics working.

```
40 PROCswap:PROCface:PROCswap:PROCface
```

Now we need to set T to the number of centiseconds that have elapsed since 12 o'clock. We also need to set HOURS so that it is the relevant distance between the hours. The minute and second hands will jump a minute and a second at a time, respectively.

```
50 T=seconds*100+minutes*6000+hours*360000
60 hours=T/360000
```

Next we need to draw the hands in. We also need to keep a copy of where they are so that we can remove them later. We can then swap so that the face plus hands is visible and wait for the user to press a key to synchronise the clock.

```
70 GCOL0,1:PROChands(hours,minutes,seconds)
```

```
 80 s=seconds:m=minutes:h=hours
 90 PROCswap:A=GET
```

Now we must start the clock by setting TIME to T. For timing we will wait until TIME crosses the hundred border. For this purpose we need to keep a copy of what TIME DIV 100 equals at this point in t, so that when we are waiting to display the next second we can wait until TIME DIV 100 doesn't equal t.

```
100 TIME=T
110 t=TIME DIV100
```

Now as we are displaying the time given by t we need to set up the time given by t + 1 in the other block. To do this we must first remove the previous hands which are located at s, m and h. Then we must set these variables to the current hand positions ready for the next move.

```
120 GCOL0,0:PROChands(h,m,s)
130 s=seconds:m=minutes:h=hours
```

If TIME has passed 4320000 (12 hours in centiseconds) then we need to set it back 12 hours.

```
140 IFTIME>4319999 TIME=TIME-4320000
```

Next we must set up the new hands.

```
150 seconds=(t+1)MOD60
160 minutes=(t+1)DIV60 MOD60
170 hours=(t+1)/3600
```

However, the + 1 for the hours practically makes no difference (it moves the hour hand on by 0.00027 of an hour) so we can ignore it. We are now ready to draw the new hands on the concealed screen.

```
170 hours=t/3600
180 GCOL0,1:PROChands(hours,minutes,seconds)
```

Now all we need to do is wait until the second is up

and swap screens before repeating the whole process.

```
190 REPEATUNTILTIME DIV100<>t
200 PROCswap:GOTO110
```

One word of warning − because all printing is concealed, any error messages will never appear. If there is an error the machine will just appear to stop working. To get around this while you are typing in and debugging the program, try adding this line:

```
1 ONERRORMODE7:REPORT:PRINT" at line
  ";ERL:END
```

Three-dimensional graphics

Computers are being used increasingly now for producing three-dimensional graphics, particularly on television. In the home micro market there are an increasing number of three-dimensional games. In passing, it would be useful to take a look at some of the simple mathematics behind this subject.

We are faced with the problem of converting a set of three-dimensional coordinates into two-dimensional ones for plotting on the screen. We know from experience that an object further away appears smaller and we would probably guess (correctly, as it happens) that if an object goes twice as far away as it looks half the size. Thus we could form a rule that apparent size is inversely proportional to distance.

Let's assume that we have three-dimensional axes X, Y and Z, where X and Y are horizontal and Z is vertical. Let's imagine that the origin is about 2 metres off the ground and that we place a camera at this point looking vertically downwards. Let's further assume that we have a square of cardboard.

We place this horizontally at different levels with its centre always on the Z axis, photographing each position. Can we make a rule about where the corners will appear in each photograph? We can assume that the image on the photograph will still be a square. If we place a drawing horizontally under the camera and photograph it we would

expect to get a precise copy of the original drawing. It might be bigger or smaller depending on what level we placed the drawing at.

From this we can guess that for one value of Z coordinate, given the X and Y coordinates of a point on that Z plane, the X and Y coordinates of the point's image on the photograph, A and B, would be given by:

$$A = k X$$
$$B = k Y$$

We already know that apparent size is inversely proportional to the distance from the camera to the point, so as the camera is at the origin and we can take the Z coordinate of the plane as this distance, this suggests that A and B are given by:

$$A = k X/Z$$
$$B = k Y/Z$$

The constant k will be a scale factor − the power of the lens we use, perhaps. As any point under the camera can be described as a point on a horizontal plane, we have here the equations for converting the 3D point into a 2D image.

If you didn't follow all that, don't worry. All you need to remember is that by looking along the Z axis from the origin you can find the image of a point by using these equations.

As an example, let's try drawing a cube on the screen. The coordinates of the corners of this cube will be (-10,10,40), (-10,-10,40), (10,-10,40), (10,10,60), (-10,10,60), (-10,-10,60) and (10,-10,60). First we need a procedure that will do the job of the PLOT command, but in 3D.

```
10000 DEFPROCplot(P%,X,Y,Z)
10010 PLOTP%,X*S/Z,Y*S/Z
10020 ENDPROC
```

Notice that the scale factor S will need to be defined at the beginning of the program. We can now draw our cube by drawing the nearest surface, then the

four edges joining the front surface to the back surface, then the back surface itself.

```
 10 MODE4:VDU23,1,0;0;0;0;29,640;512;
 20 S=1000
 30 PROCplot(4,10,10,40)
 40 PROCplot(5,-10,10,40)
 50 PROCplot(5,-10,-10,40)
 60 PROCplot(5,10,-10,40)
 70 PROCplot(5,10,10,40)
 80 PROCplot(5,10,10,60)
 90 PROCplot(4,-10,10,40)
100 PROCplot(5,-10,10,60)
110 PROCplot(4,-10,-10,40)
120 PROCplot(5,-10,-10,60)
130 PROCplot(4,10,-10,40)
140 PROCplot(5,10,-10,60)
150 PROCplot(5,-10,-10,60)
160 PROCplot(5,-10,10,60)
170 PROCplot(5,10,10,60)
180 PROCplot(5,10,-10,60)
190 END
```

This produces a recognisable cube but is not the most exciting piece of art. It would be better if we could look at the cube from a different angle. However, our 3D equations will not let us do this. The solution, of course, is that if the mountain can't come to Mohammed then Mohammed must go to the mountain − we have to rotate the cube.

To do this we need to look at the method of rotating a 2D point about the origin. Those of you who have dabbled at all in matrices will know that the general rotation matrix about the origin, by an angle A, is:

$$\begin{pmatrix} \text{COS A} & \text{- SIN A} \\ \text{SIN A} & \text{COS A} \end{pmatrix}$$

For those of you who have not experienced the joys matrices this will mean precisely nothing! But all you need to know is that if we have coordinates (X,Y) and we wish to rotate them about the origin by an angle A anti-clockwise, to new coordinates (P,Q), then P and Q can be calculated using the two

formulae:

P = X * COSA - Y * SINA
Q = X * SINA + Y * COSA

In thee dimensions we can adapt these formulae by rotating about, say, the X axis. This means that the X coordinate of the point stays the same and we can use the two formulae above, except that we use Y and Z, to rotate the other two coordinates. So for our example program the PLOT routine becomes:

```
10000 DEFPROCplot(P%,X,Y,Z)
10010 Q=Y*COSA-Z*SINA
10020 R=Y*SINA+Z*COSA
10030 PLOTP%,X*S/R,Q*S/R
10040 ENDPROC
```

However, this rotates around our viewpoint. This doesn't help much. We really need to shift the coordinates so that the centre of the cube is over the origin, then rotate the cube about the X axis, and then shift the cube back again so that we can view it.

```
10000 DEFPROCplot(P%,X,Y,Z)
10010 Z=Z-DR
10020 Q=Y*COSA-Z*SINA
10030 R=Y*SINA+Z*COSA
10040 R=R+DR
10050 PLOTP%,X*S/R,Q*S/R
10060 ENDPROC
```

We also need to set the values of A and DR at line 20.

```
   20 S=1000:A=-PI/5:DR=40
```

Let's now look at a more complicated example − drawing a cup or wine glass. This is not as difficult as it sounds, as a cup has rotational symmetry. This means that we need only store data for half of a slice through the axis of symmetry of the cup. We can then rotate this a number of times about the axis of

symmetry and join all the adjacent points.

The first thing is our 3D procedure.

```
1000 DEF PROCplot(P%,X,Y,Z)
1010 P=Y*C-Z*I
1020 Q=Y*I+Z*C+D
1030 PLOTP%,X*S/Q,P*S/Q
1040 ENDPROC
```

Notice that, because the sine and cosine of the angle of rotation about the X axis remain constant as the angle remains constant, we can define these at the start of the program as I and C, and hence save a lot of calculation. Also, the initial coordinates will be centred on the origin so we can rotate these straight away. We then need to add a constant to the Z coordinate to shift the point away from the origin before we can view it.

The beginning of the program will need to look like this:

```
10 MODE1:VDU23,1,0;0;0;0;
20 VDU29,640;195;19,3,2;0;
30 D=300:S=2400
40 I=SINRAD240:C=COSRAD240
```

The first piece of data for the cross-section of the cup will need to be the number of points used to describe it. Because we will be joining the first point in this description to the second, we will need to keep two adjacent points in variables at one time. We can read in the number of points and the first point. The number of lines will be one less than the number of points. Each line will need a colour associated with it.

```
50 READN,R,Z:FORM=2TON
60 READR1,Z1,C%:GCOL0,C%
```

Before repeating the loop we will need to copy R1 and Z1 into R and Z. Now we are ready to rotate the line formed by these two points around the Z axis. At each step we must joint the previous second

point to the new second point and join the two new points together.

```
 70 P%=4:FORA=0TOPI*2.01STEPPI/20
 80 PROCplot(P%,R1*COSA,R1*SINA,Z1)
 90 PROCplot(5,R*COSA,R*SINA,Z)
100 PROCplot(4,R1*COSA,R1*SINA,Z1)
110 P%=5:NEXT
120 R=R1:Z=Z1
130 NEXT
140 END
```

Finally we need the data.

```
150 DATA7,0,-5,32,-5,3,32,0,2,8,10,2,8,
        35,1,24,40,1,24,85,3
```

This is, of course, only a simple example of 3D graphics.