17,000+ Digitally Remastered BBC and Electron Pages.

The 55 BBC Micro Books CD-Rom was released in 2013 to critical acclaim.

It's continued to be one of our best selling products.

That's why we thought we'd make an improvement by enlarging the collection to 80 books.

That's over 17,000 pages of high quality viewing (and printing) in popular formats for most computing platforms. All on one CD-Rom*.

Owners of MS Windows and Macintosh machines can access the books in Adobe PDF and HTML format. As can RISC OS users who also will benefit from the original, Impression and EasiWriter files.

Plus over 3,000 typed-in and debugged listings, ready to run.

Many of these programs will work on modern RISC OS computers like the Raspberry Pi.

That's because they're written in BBC Basic which has come built in to every Acorn or RISC machine since the BBC Model A.

Programs will also run much faster on RISC OS, for example 3D graphics routines produce instantaneous results.

DFS disc images of the programs are supplied for BBC emulators or for writing to physical media for use with 'real' machines.

What we haven't enlarged is the price. It's still just £14.00** on CD-Rom*.

Whether you are a student learning to code, a professional or hobbyist user or just a collector, don't miss out on this astounding compilation of 80 BBC and Electron books.

100 Programs for the BBC
100 Programs for the BBC 100 Programs for the Acorn Electron
21 Games for the BBC
21 Games for the Electron
35 Educational Programs for the BBC Micro
36 Challenging Games for the BBC Micro
40 Educational Games for the BBC Micro
*40 Educational Games for the Electron
60 Programs for the BBC Micro
*60 Programs for the Electron
Advanced Basic Rom User Guide
Advanced Graphics on the BBC Model B
*Advanced Graphics on the Acorn Electron
Advanced Machine Code Techiques
Advanced Programming for the BBC Micro
Advanced Programming Techniques for the BBC Micro
Advanced Programming Techniques for the Electron
Advanced User Guide for the Electron
*Adventure Games for the BBC Micro
*Applied Assembly Language on the BBC Microcomputer
The Basic ROM User Guide
The BBC Micro Book
BBC Micro Graphics and Sound
BBC Micro Expert Guide
*BBC Micro and Electron Book
*The BBC Micro Gamesmaster
*The BBC Micro Rom Book
*BBC Micro Wargaming
*BBC Programs Volume 1
The BBC Micro Revealed
Best of PCW Software
*Biology Programs for the BBC Computer
Brainteasers for the BBC and Electron
*Building Blocks for BBC Games
*Cracking the Code on the BBC Micro
Creating Adventure Programs on the BBC Micro
Creative Animation and Graphics on the BBC Micro
Creative Assembler How To Write Arcade Games for the BBC and Electron
Creative Graphics on the BBC Micro B
*Discovering BBC Micro Machine Code
*Drawing Your Own BBC Programs
*Educational Games for the BBC Micro
*The Electron Book
*Electron Programs
*Electron Graphics and Sound
Essential Maths on the BBC and Electron
Games and Other Programs for the Electron
Games BBC Computers Play
The Electron Gamesmaster
Giant Book of Arcade Games
Graphic Art for the BBC Computer
Graphics on the BBC Microcomputer
Graphics Programming on the BBC
Graphito
Graphs and Charts on the BBC Microcomputer
Handbook of Procedures & Functions
How to Write Adventure Games on the BBC and Electron
Instant Arcade Games for the BBC Micro
*Instant Arcade Games for the Electron
*Invaluable Utilities for the BBC Micro
Invaluable Utilities for the Electron
The BBC Micro Machine Code Portfolio
Making Music on the BBC Micro
Mastering Assembly Code
*Mastering Interpreters and Compilers
Microguide for the BBC
More Virgin Games for your BBC
BBC Micro Music Masterclass
PCW Games Collection for the BBC
Practical Programs for the Electron
*Procedures and Functions in BBC Basic
BBC Micro Programs in Basic
Quality Programs for the BBC
*Quality Programs for the Electron

* Available on USB flash drive for £2.00 supplement.
** Prices correct at April 2021. Upgrade from the '55 Books' is £14.00 (i.e. same price)

To order visit www.dragdrop.co.uk (Paypal) or email sales@dragdrop.co.uk for details of internet bank payments. E&OE

37 CHANGING VECTORS

The signalmen at Victoria station in London can arrange for a train arriving on any track to arrive at any platform. Railways invariably have an up line and a down line; one deals with input, the other with output. This is the idea of vectoring. In mathematics a vector is a quantity that can have both magnitude and direction. In computing, a vector is a location containing an address in the operating system which can be changed so trapping any program which is using it. Figure 37.1 shows the concept.

The system vectors are stored in RAM between &200 and &236 hex. To continue the analogy with railways, this block of memory is like a signal box. You can experiment by pulling one of the levers, but if you do not know what you are doing you could easily send the Orient Express into a siding, or worse.

The system vectors are as complicated as a signal box; you can crash the computer by altering them. If this occurs, you may be forced to switch the machine off, thus losing any program or data you happen to have in memory. As a general recommendation, do not do anything which might interfere with the normal operation of the machine; the interrupt vectors in particular should be left alone by the novice.

You can begin experimenting by changing the input and output vectors. Suppose you want to change all characters typed at the keyboard to lower case; you could do it by trapping the input vector, RDCHV.

Here is what you do to change vectors. First, put the operating system vectors and the addresses pointed to by the vectors into variables like this:

```
REM Input vectors: ------------
REM
RDCHV%=&210:REM address of Read Character Vector
RDCHVL%=?RDCHV%:REM store low byte
RDCHVH%=RDCHV%?1:REM and high bytes

REM store address pointed to by Read Character Vector:
OSRDCH%=RDCHVL%+256*RDCHVH%

REM
REM Output vectors: -----------
REM
WRCHV%=&20E:REM Address of Write Character Vector
WRCHVL%=?WRCHV%:REM store low byte
WRCHVH%=WRCHV%?1:REM and high byte
REM store address pointed to by WRCHV:
```

```
OSWRCH%=WRCHVL%+256*WRCHVH%
REM
```

Program 37.1 VECEX. Changing vectors

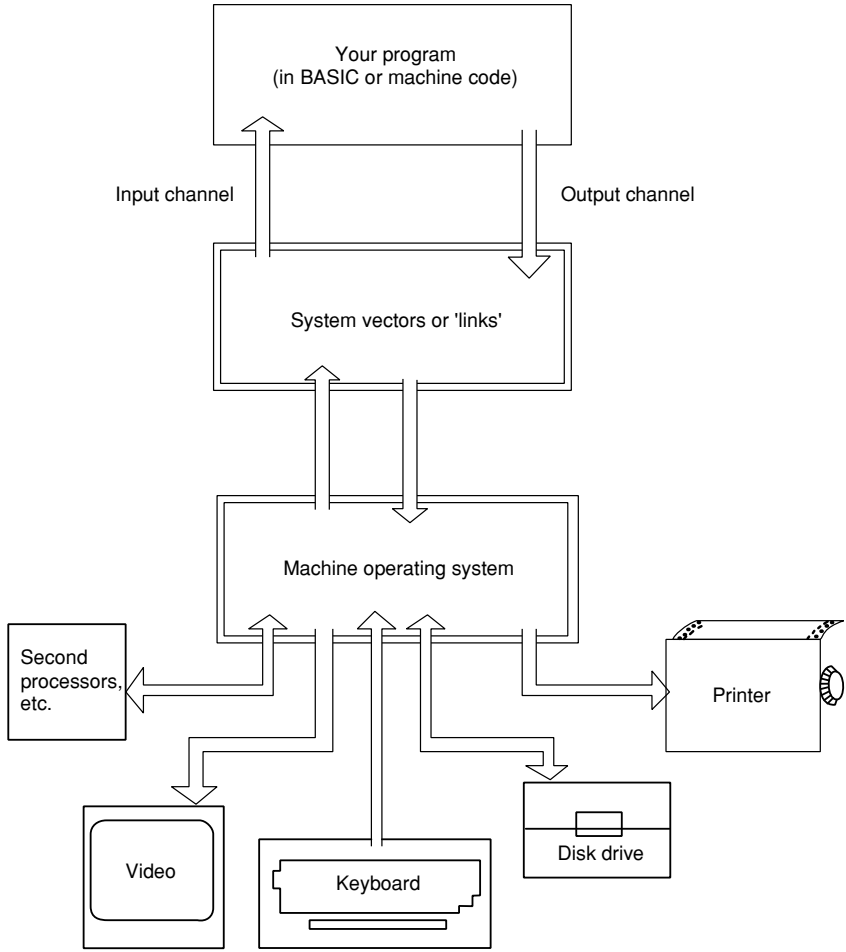


Figure 37.1 Machine operating system vectors

memory location, in this case &FE60. Also associated with it is location &FE62 which controls if the port is input or output. Each bit position in &FE62 corresponds to a bit position in the user port and if a bit in &FE62 is a 1 then the corresponding bit in the user port is an output but if it is a 0 then the bit is an input. Thus to set all the bits of the port as inputs the instructions: LDA £0: STA &FE62 and to set it to all outputs: LDA #&FF: STA &FE62. If the programs have to work using a second processor these are written LDA #& 97: LDX #&62: LDY #0: JSR OSBYTE for an input port and LDA #& 97: LDX #&62: LDY #&FF: JSR OSBYTE for an output port.

Reading the port consists of reading the memory &FE60 for example LDA &FE60. This will not work if run on a second processor, however, and under these circumstances then the OSBYTE subroutine with A set to &96 and X set to &60 must be used, the value being returned in the Y register. Thus reading the port becomes: LDA #&96: LDX #&60: JSR OSBYTE.

Similarly to write to the port using a second processor we have LDA #&97: LDX #&60: JSR OSBYTE which will write the byte contained in the Y register.

AN EXPERIMENT BOARD

In order to make any significant progress in understanding digital interfacing using the computer some hardware is necessary. Figure 10.1 gives the circuit diagram of a simple electronic experiment board, developed at the University of Salford, and this may be constructed by readers with a little electronics experience. It contains three types of output device connected to the printer port: light emitting diodes so that a visual display of all 8 bits is obtained, a power transistor on bit 0 so that higher power devices such as relays and motors may be driven and also a DIGITAL TO analog converter so that a voltage output of the digital number is obtained. It also contains 8 switches connected to the user port so that digital input to the computer can be obtained.

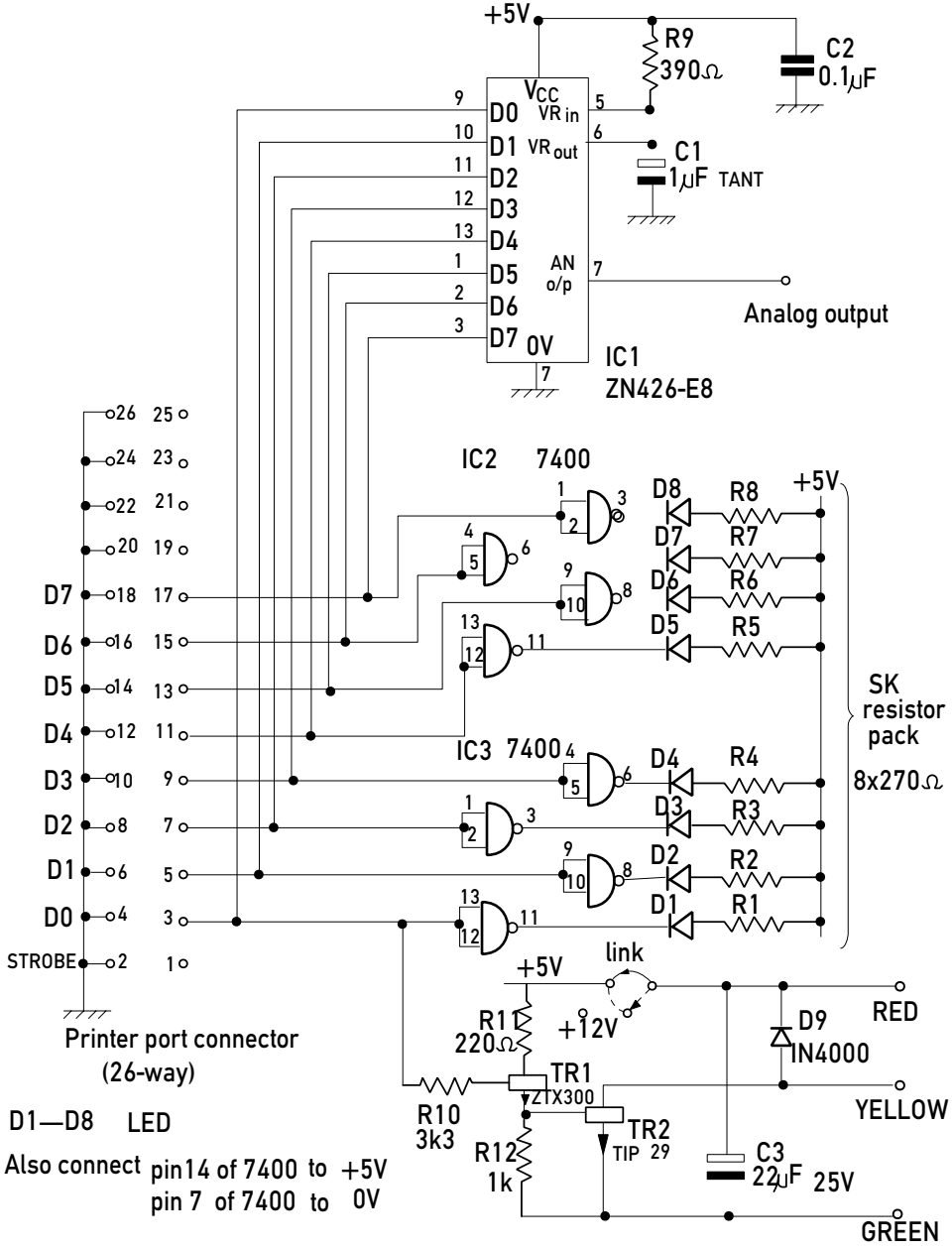

+5V
R9
390Ω
C2
0.1μF
VCC
D0
D1
D2
D3
D4
D5
D6
D7
VR in
VR out
AN o/p
0V
C1
1μF TANT
Analog output
IC1
ZN426-E8
IC2 7400
IC3 7400
D8
D7
D6
D5
D4
D3
D2
D1
R8
R7
R6
R5
R4
R3
R2
R1
SK resistor pack
8x270Ω
STROBE
Printer port connector
(26-way)
link
+12V
R11
220Ω
TR1
ZTX300
R10
3k3
R12
1k
TR2
TIP 29
C3
22μF 25V
D9
IN4000
RED
YELLOW
GREEN
D1—D8 LED
Also connect pin14 of 7400 to +5V
pin 7 of 7400 to 0V

Figure 10.1a

```
7140 PRINT TAB(1);"fall in the river."'
7150 PRINT TAB(3);"Press any key"
7155 PRINT TAB(4);" to start"
7160 IF INKEY$(0)="" THEN GOTO 7160
7170 ENDPROC
```

PROCinit

PROCinit starts off in the usual way by defining the graphics characters and colours used but in this case it is also responsible for printing the coloured strips that represent the road and the river.

```
1000 DEF PROCinit
1010 VDU 23,224,&3C,&7E,&7E,&7E,&7E,&FF,&FF,&FF
1020 VDU 23,225,&7E,&7E,&7E,&7E,&7E,&7E,&7E,&7E
1030 VDU 23,226,&7E,&7E,&FF,&FF,&FF,&7E,&7E,&7E
1040 VDU 23,227,&B9,&52,&1C,&1E,&1C,&52,&B9,&00
1050 VDU 23,228,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
1060 VDU 19,0,0,0,0,0,0:REM 0=BLACK
1070 VDU 19,1,1,0,0,0,0:REM 1=RED
1080 VDU 19,2,6,0,0,0,0:REM 2=CYAN
1090 VDU 19,3,2,0,0,0,0:REM 3=GREEN
1100 A%=32
1110 B%=32
1120 C%=32
1130 D%=32
1140 VDU 23,1,0;0;0;0;
1160 X%=2
1170 Y%=15
1175 VDU 26:CLS
1180 FOR I=0 TO 31
1190 COLOUR 128+1
1200 PRINT TAB(3,I);SPC(5);
1210 COLOUR 128+2
1220 PRINT TAB(11,I);SPC(5);
1230 NEXT I
1240 GAME_END=FALSE
1250 MAX=1000
1260 ENDPROC
```

Lines 1010 to 1050 define the cars, logs and frog. CHR$(224) is a car or lorry front, CHR$(225) is a lorry middle section and CHR$(226) is a car or lorry end. You can see the way that these three characters go together in Fig. 4.2. CHR$(227) is the frog and CHR$(228) is simply a solid block used to make up logs (see Fig. 4.2). The colours selected by

lines 1060 to 1090 are black for the background, red for the road and for the logs, blue for the river and the traffic and green for the frog. You may be surprised at the choice of bue cars on a red road and red logs on a blue river but this does simplify the game quite a lot. When the frog is crossing the road it must avoid blue cars and when it is crossing the river it must avoid blue water and so all through the game the colour blue indicates an area where the frog shouldn't go.

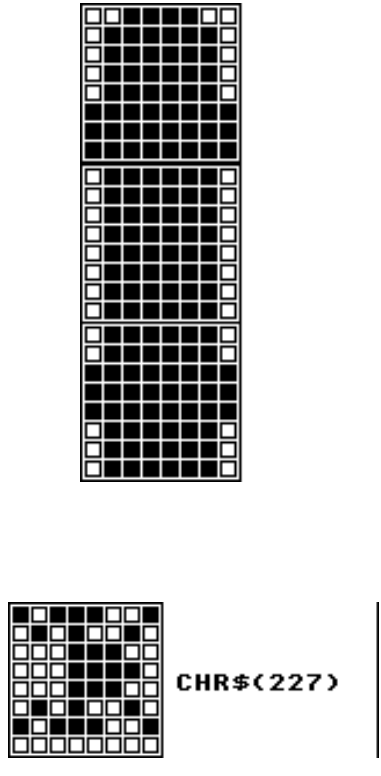


Fig. 4.2. Graphics characters for lorry, frog and log

Lines 1100 to 1130 initialise the variables A% D%, to 32, the ASCII code for space. These variables are used by PROCscroll and are best described in that section. X% and Y%, are the current co-ordinates of the frog and are initialised by lines 1160 and 1170.

The VDU 26:CLS command in line 1175 removes any text windows

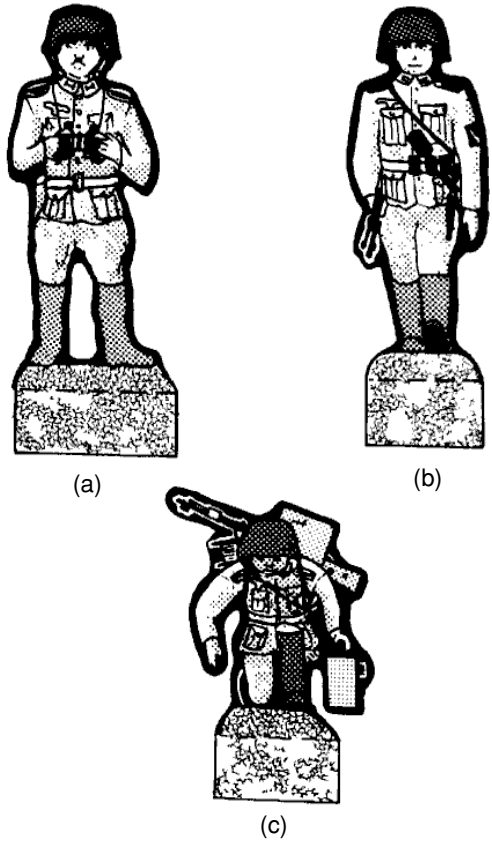


Fig. 11. Designs tor cut-out figures of the German forces in THE BRIDGE. (a) Officer; (b) Private; (c) Sapper.

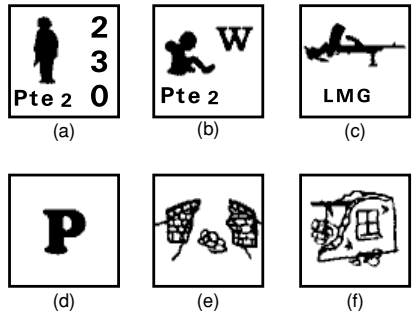


Fig. 12. Examples of counters for playing THE BRIDGE. (a) German Private; (b) reverse of (a) for use when wounded; (c) portable weapon; (d) panic marker; (e) blown bridge marker; (f) rubble marker.

All units begin the game in active status. During the game they may panic, become wounded, or be killed in action. With models, a panicking soldier may be indicated by turning the model to face away from the enemy. A wounded soldier is indicated by lying the model on its side, or by replacing it with a model in a suitable posture. With counters a panicking soldier is indicated by placing a 'panic' marker over the counter. A wounded soldier is indicated by inverting the counter. The reverse side of the counter bears a 'W' to indicate this status. Units killed in action are removed from the table or board.

Board wargamers need about a dozen 'panic' markers. A single marker is needed to indicate that the bridge has been totally destroyed, and another to indicate that it has been partly damaged. One of these is placed on the bridge hex when damage has been effected. You also need three or four 'rubble' markers to be placed on building hexes to indicate that the building has been reduced to rubble by a demolition charge. Model wargamers will be able to devise their own ways of indicating damage to the bridge or buildings.

In the rules which follow, those who are playing with models are asked to make appropriate modifications, depending on the exact scale of their models.

Up to four counters representing men may be stacked on any one hex. In addition you may stack any number of portable weapon counters. But weapon counters cannot move on their own; they must be 'carried'. One man may carry only one portable weapon (sappers carry none).

Sequence of play

(1) The Allied player deploys units (including portable weapons) first, placing them anywhere within the area bounded to the east and south by the road and to the west by the river.

(2) The German player deploys units second, placing them in any of the three columns of hexes on the east side of the map.

(3) The game consists of ten turns, each of which is divided into two player- turns, the German having the first player-turn.

(4) Each player-turn is divided into three phases, intended to represent about one minute of real time:

(a) *Advance Phase:* the player advances any or all units. Units may not enter hexes occupied by enemy units.

(b) *Fire/ Advance Phase:* the player either advances units again, or fires their weapons. Units may enter hexes occupied by the enemy. Panic markers are removed from all enemy units before firing begins.

(c) *Close Combat Phase:* in any hex in which there are units belonging to both sides.

FILL DEMO

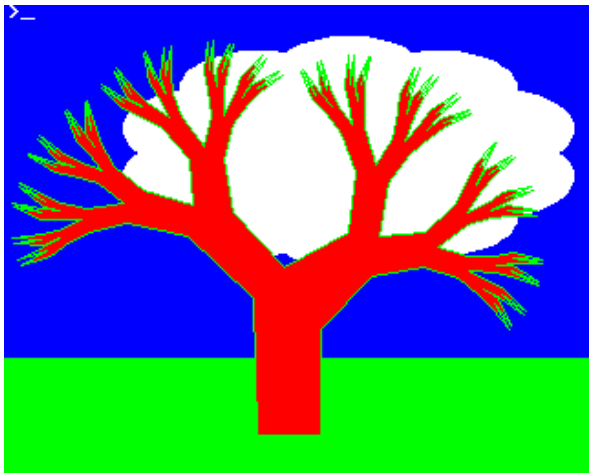
At the end of this program is the data for a machine code routine to fill in any area on a graphics screen that has a border of a given colour in any given colour. The routine will work in modes 0,1,2,4 & 5. The area it is to fill must be fully enclosed by the border colour otherwise the fill colour will escape and try to fill the rest of the screen. The data for the machine code program is read in from the data statements and checked as it is put into memory by lines 30-100.

To run the demo program PAGE must be set to &1100 and then the program CHAINed from tape. The program then puts the machine code into memory &0E00 to &1100. It then draws a cloud and a tree and uses the fill routine to fill them in.

NOTES ON THE PROGRAM

The parameters of PROCFILL are the x & y coordinates of the start position for the fill, colour in which to fill the area and the border colour of the area.

PROCTREE is a recursive routine and calls itself to draw the branches of the tree.



CHAPTER 3

GENETICS

Introduction

In many ways, the study of genetics lies at the heart of modern biology. If there is a unifying concept in the life sciences, it may by the idea of evolution by natural selection, an idea described quantitatively in the language of genetics.

Until quite recently, genetics meant simply the study of inheritance; and that meant Mendei's laws and their exceptions. This is classical genetics.

The first three programs described in this chapter fall under this heading. MENDEL is a simple simulation of some of Mendel's breeding experiments with peas. LINK simulates the genetics of a dihybrid genetic situation invoiving two pairs of alleles occupying loci on the same chromosome. SEXL simulates experiments involving a sex-linked phenotype in Drosophila.

Increasingly, genetics is being used to generate an understanding of the processes that have produced the staggering variety and range of forms of living things, of evolution. This use of genetics is often called population genetics. The fourth and fifth programs in this chapter are simulations in this area.

SICKLE allows the user to investigate a simple model in which selection brings about the evolution of a model population. DRIFT demonstrates that in small popuiations, evolution can occur in the absence of selection.

MENDEL – a program to illustrate classical genetics

This program is a simulation of experiments to demonstrate and investigate Menders Laws.

Biological Background

Mendel's laws are the cornerstone of genetics. Students of O and A level biological sciences are expected to gain a thorough understanding of these laws. Future study in genetics is really impossible without an easy familiarity with them.

Mendel's laws are the rules of simple inheritance. The first law concerns the particulate nature of inheritance, and states that for each feature of an organism (phenotype), each somatic cell carries two particles or alleles.

Further, the law states that each gamete produced by an organism has only one of these alleles. The nature of the two alleles in somatic cells, is referred to as the cells' genotype. It is often the case that the two alleles are not identrcal, and one may be dominant to the other.

One of Mendel's experiments involved crossing together two sorts of pea plants, one with very tall stems and one with short or dwarf stems. He found that if the parent plants used in this cross came from pure strains of tall or dwarf, that the progeny or offspring produced, the so-called F1 generation were all tall stemmed. Further, he found that if these F1 tall plants were allowed to self-fertilise, to intercross, then the progeny produced consisted of 75% tall plants and 25% dwarf ones.

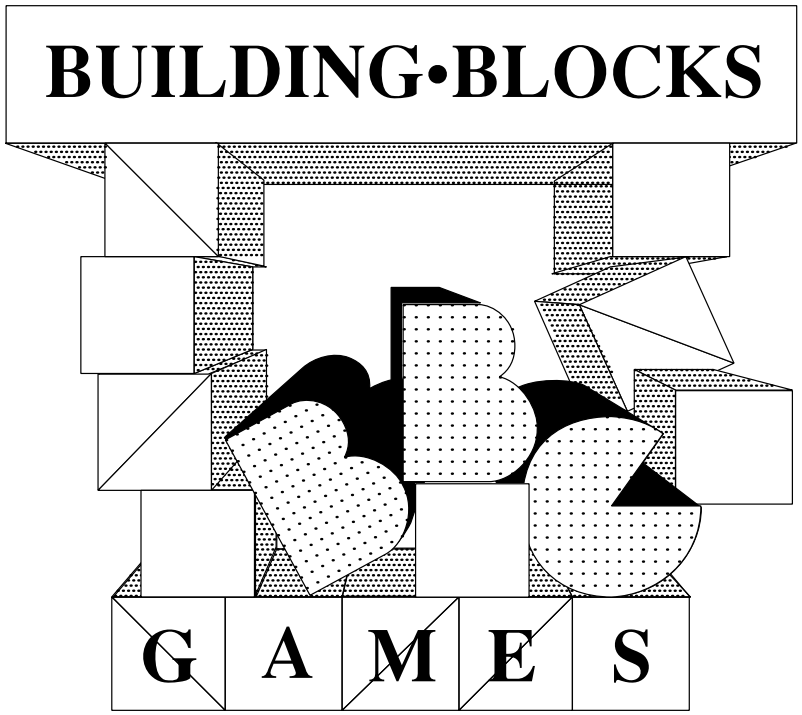
The information for dwarf had 'skipped' a generation, but must have been present in the F1 tall plants; it had been masked by tall information. Tall is dominant, dwarf recessive.

This result can be explained if it is assumed that each parent cell, except gametes contain two alleles. Let T be the tall allele, and t be the recessive allele. Then, the tall parent has genotype TT, and the dwarf parent has genotype tt. If we assume that the gatnetes produced by these parents each contain one of the pair of alieles in somatic cells, then each tall gamete has genotype T, and each dwarf genotype t. Thus each fertilisation of a tall gamete by a dwarf one {or vice versa) gives rise to a zygote and hence a new plant of genotype Tt.

The phenotype of this plant is tall. Now, if one of these plants is allowed to self-fertilise, or is crossed to a plant of the same genotype, then half of the gametes produced by the F1 hybrid have the genotype T and half the genotype t. Assuming that fertilisation occurs at random, and that gametes containing the allele T are as likely to be fertilised, or effect fertilisation as those gametes containing t, then the probabilities of the production of zygotes containing at least one T allele, or no T alleles are .75 and .25 respectively. A similar argument applies to the simultaneous and independent tossing of two coins. The probability of a result consisting of a least one head is .75. That of two tails is .25.

Mendel's first law allows a prediction of the outcome when an F1 hybrid ts crossed to an individual with the genotype tt. (An individual with both alleles the same is homozygous for that pair of alleles, else it is heterozygous) When a heterozygote, Tt is crossed with a homozygous recessive, it in this case, Mendel's first law predicts that the progeny should consist of 50% tall and 50% dwarf plants. The probabilities of occurence of zygotes containing at least one T or no Ts are respectively .5 and .5. These probabilities are obtained, as above, by multiplying together the relative frequencies of gametes of different genotypes produced by the two parents. In the present case, the F1 plant produces two gamete types, T

BUILDING•BLOCKS



MELBOURNE HOUSE

It is good practice to build up a library of tested procedures, either before you write a program or build them up collectively from programs written. To make full use of a procedure library, the procedures need to be 'transportable'. This means that a procedure used in one program should be able to be used within another program without modificaton.

This sometimes requires the use of global variables. A global variable is a variable whose contents may be used or changed anywhere within a program (unlike a LOCAL variable). Take, for example, the common procedure "clear_message" used in this book.

The procedure's variables (b, f, x, y, x2, y2) are initialized to their default values in the initialization section of a program. If you wanted to execute the "clear_message" procedure with the b variable (background colour), set it to 3 and the rest of the variables are left at their default values. You would use the statement

```
PROCclear_message(b, t, x, y, x2, y2)
```

As long as these procedure variables are initialized, this procedure can be said to be transportable.

BUILDING A PROGRAM FROM SCRATCH

When you write a program using PROCs and a main-loop the first step is to break your program down into the steps that it will actually go through.

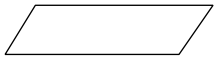
A flowchart of your program would really help as you will identify the different PROCs you will need from the flowchart.

Because we are aiming to design our programs around main-loops and procedures, we will keep this concept in our flowcharting too, even to the point of using REPEAT-UNTIL in the charts.

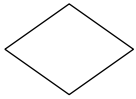
A flowchart is a way of representing a process as a collection of steps to be taken and the effect of decisions on further steps within the process. When flowcharting we use the following symbols.

denotes an entry or an exit point

denotes a process



denotes input or output



denotes a decision

Calculate YB'

In *figure 2.11* we need to find the length ON to discover the new coordinate YE'. It is easy to see that the angle YOY' is equal to XOX" and also equal to B'PR according to the same rule above.

Also:
PB' = XB
OP = YE

and:
OQ = OP COS(a)
B'R = PB' SIN(a)
NQ = B'R
ON = OQ + QN

ON = OP COS(a) + PB' SIN(a)

Therefore:-

YB' = XB SIN(a) + YE COS(a)

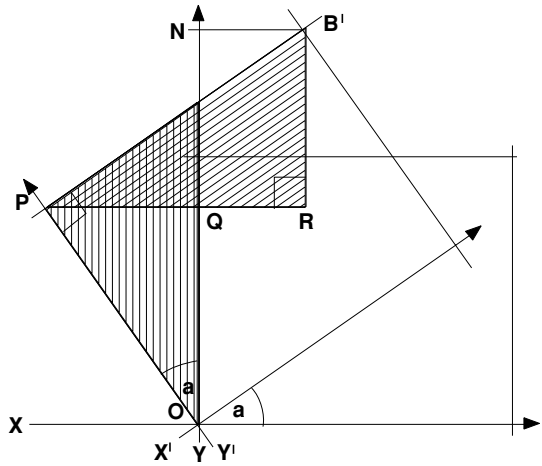


Fig. 2.11.

We generalize the relationship derived above to apply to any point with coordinates X and Y to obtain its coordinates after rotation, represented by U and V in the formulae below.

U = X COS(a) – Y SIN(a)
V = X SIN (a) +Y COS(a)

You will note that a, the angle of rotation, is measured anticlockwise. It is important to remember that the BBC computer works in radians not degrees when dealing with COS and SIN (one radian is about 57 degrees).

We are now in a position to write a procedure to rotate any shape through a given angle.

```
13000 DEF PROCROTATE(A)
13002 LOCAL PC,U,V,C,S
13004 C=COS(RAD(A))
13006 S=SIN(RAD(A))
13008 FOR PC=1 TO NOR
13010    U=P(PC,2)*C-P(PC,3)*S
13012    V=P(PC,2)*S+P(PC,3)*C
13014    P(PC,2)=U
13016    P(PC,3)=V
13018 NEXT PC
13020 ENDPROC
13022 :
```

We will now amend our program to rotate the square. To avoid confusion, first delete the lines that are no longer needed, and type in the lines below:

```
DELETE 190, 240
  190 PROCSCALE(3,3)
  200 PROCTRANS(640,512)
  210 PROCDRAW
  220 PROCROTATE(5)
  230 PROCDRAW
  240 :
```

```
10030 PRINT''"Unfortunately, the main computer has
"'"just gone mad and will not monitor the"'"cooler
s automatically. It will monitor"'"them, one at a
time, if you give it an"'"instruction to do so, bu
t it will only"

10040 PRINT"accept your right to tell it what to d
o if you can answer questions that it putsto you."

10050 PROCcont(23):CLS

10060 PRINT'"Your task is to keep the coolers work
inguntil the repair team have fixed the"'"computer
."''"All the questions you will be asked are answe
red by giving the name of an"'"Element."

10070 PRINT'"First of all you will be told the sym
bolused to stand for the element. If you"'"get the
 answer wrong you will be given asecond clue - the
 Atomic Weight of the  element. If you still give
the wrong"

10080 PRINT"answer you will be given a final clue
- the Atomic Number of the element."

10090 ENDPROC
```

Main Program

It is interesting, perhaps, to try to establish where a game idea comes from. In this case I wanted to use the chemical elements as data for a clue-giving game with a similar structure to King of the Castle. I thought that a picture of an atom, with electrons in orbit round a nucleus, would make a good display and would be suitable for data that concerned atomic number and atomic weight. Next I needed to decide what was to happen in response to right or wrong answers, and it seemed obvious to think in terms of the electrons being made unstable by a wrong answer, which could be illustrated by making them flash, and being restabilised by a right answer. Of course, if all the electrons become unstable then the atom must blow up (violence again!), but how do I get the human player into the scenario? Defusing a bomb? Controlling a power plant? I opted for the latter and made the atom display board in its control room, with each electron representing a cooler to keep the core under control. Now, how to work in the asking of the questions? Obvious! My old friend

Screen display from 'Power station'

the family computer. The end result is a simple, if somewhat surreal, game which is set ten removed from reality as usual.

Quite a lot of the non-graphic routines in this program are similar to those in King of the Castle, but the graphics are all different. So if you are starting with the King of the Castle program and editing it, delete lines 3000-3670 now. Lines 100, 500, 530, 540, 550, 620, 350 and 4170-9990 are the same as in King of the Castle. Lines 510, 550, 570, 590, 300, 310 and 330 all need to be changed a bit, and lines 190, 582 and 352 must be added.

In PROCinit, delete the old lines 1010-1100 and enter the new lines 1010-1080. This is a very noisy program at times because lines 570 and 582 have SOUND commands in them that use ENVELOPE1 (line 1060) to generate warning sirens when any of the coolers are overheating. If you do not like sound then leave the SOUND statements out, but it you only want it to be quieter change the last two numbers in ENVELOPE1 (now given as 150, 100) to 80, 90.

```
100 MODE2:HIMEM=H%:PROCinit
190 PROCdisable
500 FOR K%=0 TO 9%
510 PROCchoosedress
```

To produce the desired effect we first draw at each Z ordinate a wide bar aligned along the X axis of the graph. This bar is then filled with colour. Next we can draw a wide bar at the same Z ordinate but aligned along the Y axis. To complete the pseudo-solid bar we draw in a diamond shaped top for it. The result is vertical bars with different coloured sides and atop ina third colour. Mode has been used with the tops of the bars drawn in white. By changing the palette the colours could be set to any desired combination to produce a suitable artistic effect. Mode 2 would permit the use of more colours but resolution is rather poor in Made 2.

Circular three axis plots

The three axis plots we have produced so far have used linear axes but we can produce a rather interesting variation in which the base plane of the plot is circular. This type of three axis plot can create some rather attractive patterns.

```
100 REM Circular 3D plot program
110 MODE1
115 REM Set up scaling factors
120 K=RAD(180/2000)
130 M=0.707
135 REM Plot graph
140 FOR X=-100 TO 100 STEP 2
145 REM Set Y sweep limits
150 YM=5*INT(SQR(10000-X*X)/5)
160 FOR Y=YM TO -YM STEP -5
165 REM Calculate Z ordinate
170 LET Z=FNA(SQR(X*X+Y*Y))-M*Y
180 IF Y=YM THEN 200
185 REM Hidden line removal
190 IF Z<Z1 THEN 220
200 PLOT69,640+4*X,500+INT(2*Z)
205 REM Update highest Z value
210 Z1=Z
220 NEXT Y
230 NEXT X
240 END
245 REM Define Z function
250 DEF FNA(Z)=10*COS(K*(X*X+Y*Y))
```

Fig. 10.9. Program to produce a circular three dimensional plot.

To give an impression of depth the X – Y plane is displayed as an ellipse and looks like a flat disk viewed from an angle. The Z ordinates have the effect of producing a series of elliptical ridges in the disk surface. The X – Y plane in this type of plot is produced by using the squares equation for a circle as the relationship between X and Y. To produce an ellipse the X scale factor is made larger than the Y scale factor. Z values are simply added to the calculated Y co-ordinates and points are plotted at the tops of the Z ordinates. Figure 10.9 shows a program listing to produce this type of three dimensional plot. The result on the screen will be similar to Fig. 10.10.

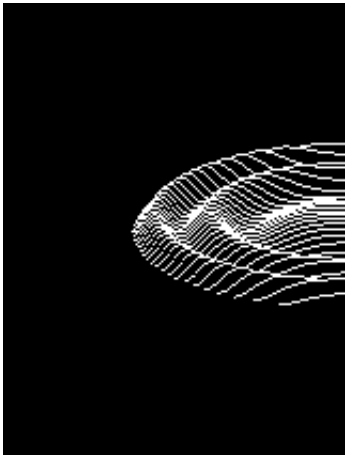


Fig. 10.10. Display produced by three dimensional circular plot program.

The shape of the plot is determined by the function used to calculate Z which is defined at the erid of the program by the DEF FN statement. A point to note here is that for the Electron the DEF FN statement should be placed after the END statement of the program. Programs of this type written for other computers will probably have the DEF FN statement near the start of the program. If such programs are being converted for use on the Electron it ts advisable to move the DEF FN statement to the end of the program as we have done here.

Line 190 in the program provides a simple form of hidden line removal which effectively blanks out any points on the curve that lie behind the front surface of the plot. For the first point in each set of Y

CHAPTER FIVE

LANGUAGE? – WHAT LANGUAGE?

Having spent all of the last chapter explaining the inner workings of Grafrite, our interpretive language, I now confess (wait for it) – Grafrite is not a language! Well, not yet. Why? Consider what happens when you press BREAK – you return to BASIC. Press the BREAK key while in BASIC and what happens? You stay in BASIC. For Grafrite to fully make the transition from a fancy piece of machine code to a language it must react as BASIC does to a BREAK: in short, stay in Grafrite.

There is another aspect to consider. Grafrite is entered via a CALL command. CALL performs a JSR to the specified address, leaving an RTS address on the stack. A full-bodied language would not be expecting to perform an RTS and as such should itself take control of the stack.

From the debugging and testing point of view, it is much better to approach language writing as outlined already. The two changes above should be left till the last. Taking control of the BREAK key means you must also take control of the error-handling facilities. Use those already available from BASIC rather than waste time and effort supplying om own routines – which could themselves be wrong!

Handling BRK

Before tackling the BREAK key let us first examine how we can trap the software BRK. In BBC BASIC the BRK instruction (opcode &00) is used to define an error. For instance, when an unknown command occurs, BASIC jumps to a suitable error message, for example 'Mistake'. The error message is stored like this:

```
BRK                 \  software BRK
04                  \ error number
"Mistake"           \ ASCII message
BRK                 \ Software BRK
```

To see this in action try the following short program:

Listing 5.1

```
 10 REM Error Messages
 20 REM Mastering Interpreters
 30 REM and Compilers
 40 REM (C) Bruce Smith 1986
 50 :
 60 P%=&900
 70 [
 80 .error
 90 LDA #7
100 JSR &FFE3
110 BRK
120 ]
130 ?P%=5
140 P%=P%+1
150 $P%="Not a Mistake!"
160 P%=P%+LEN($P%)
170 [
180 BRK
190 ]
200 CALL error
```

When you run this it will print out the error message 'Not A Mistake!' Try typing:

PRINT ERL

and the error number will be printed. So it does work. Before indirecting through BRKV the MOS does some housekeeping along these lines:

```
STA &FC             \ save accumulator
PLA                 \ set flags at BRK
PHA                 \ resave
AND #&10            \ was it an IRQ?
```

basic unit (one hexagon and one diamond) is made up of 596 mode 5 pixels, excluding the lines defining the shapes. The program simulates the seeping light by making 20 passes through each unit, filling in a few more pixels each time.

Unfortunately, the decision as to which pixels should be filled in on which pass is almost impossible to program. The incoming colour must satisfy lots of rules which are not very easy to specify. It must, for instance, start by affecting the far side of each unit, and leave the near side until quite late. While it is theoretically possible to program all such constraints, and then let the computer randomly choose pixels which satisfy them, there are two reasons for not doing it this way. First, it would take the programmer too long to appreciate and specify all the constraints; and second, it would take the computer too long to do it would spend more time discarding unsuitable pixels than it would filling in the new colour.

So I have decided for myself how many pixels should be affected in each pass, and even which ones they are. That information must be passed to the program as data. I apologise for the quantity of it, but assure you that the effect is worth the time you will spend typing it in and proof reading it. (Of course if you have the cassette of the programs, there's not a lot of effort involved.)

By a historical accident, Hexagons was first written as two programs (the accident was that it was written for the model A BBC micro, which doesn't have enough storage to run it as one program.) Although the Electron could easily handle it in one go, I have left it as two programs because of two interesting points which this raises. First there is the fairly minor matter of one program automatically calling another, with the CHAIN command. And second there is the fact that the first program stores information for the second to use, in such a way that it isn't destroyed when the second program is loaded.

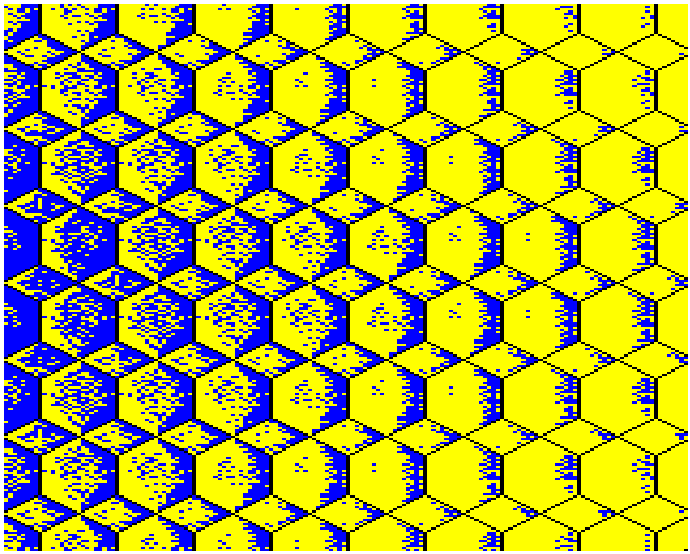
How to use the programs

Type in the first program and save it on tape (or disc, if you have one). Type in the second program and save it. You don't have to use the names that I've used for the files, but you must be sure that

the name you use in the CHAIN command in the first program is the same as the name you use for the file the second program is SAVEd in. If you have a Model A, don't type the comments in the second program - they take up too much space.

Rewind the tape to a spot before the first program, LOAD it and RUN it (or simply CHAIN it). It will search for the second program on the tape when it is ready.

The program will run until broken into with ESCAPE or BREAK. It takes about 20 minutes to complete a full cycle. I see it as fulfilling the same sort of function as a goldfish bowl — something soothing to look at now and then, rather than something to concentrate on.

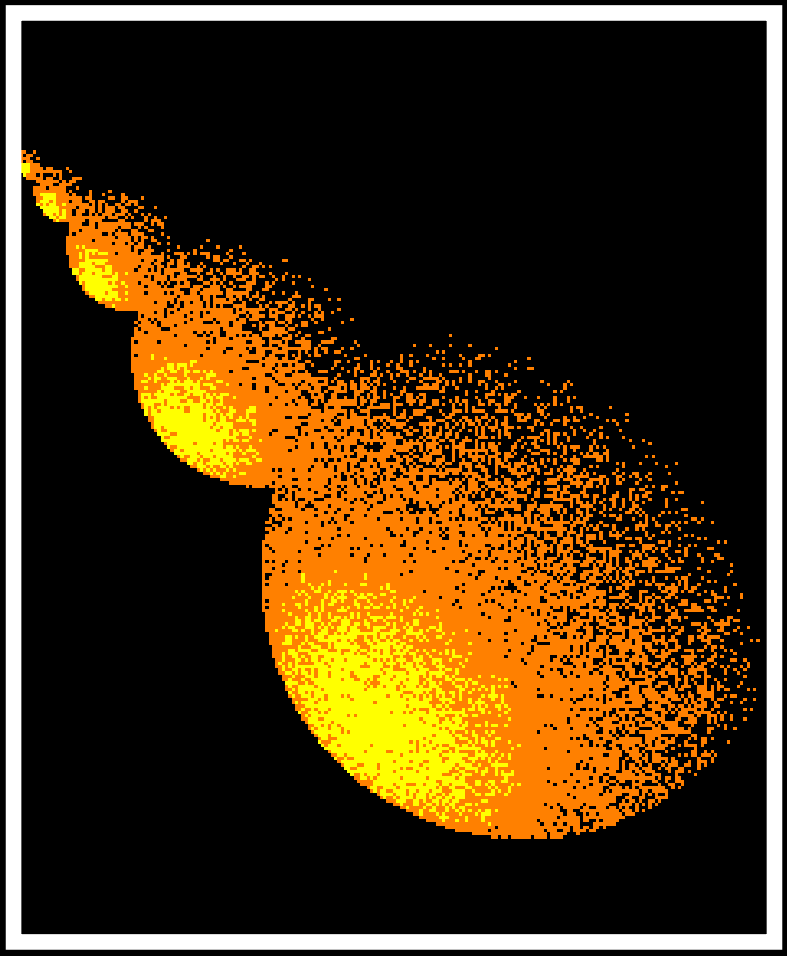


Hexagons, as a darker colour diffuses in from the left

Program listing

```
  100 REM HEXAGON - the first phase of the hexagons program
- by Simon.
  110 DATA6,0,6,34,8,35,15,19,15,23,16,9,16,12,16,14,16,16,1
6,22,17,18,17,19
  120 DATA5,34,6,35,7,0,7,34,13,12,13,15,14,10,14,14,15,11,1
```

Advanced Graphics with the Acorn Electron



IO Angell and B.J. Jones

30th Anniversary Digitally Remastered Edition

a video game. To understand fully the effect of this program it may be necessary to revise the section relating to screen memory arrangement given in chapter 5.

Example 13.2

We can also use machine code to move parts of a picture about the screen. The simple program in listing 13.3 creates a routine (placed at locations starting at &B00) that transfers 1K of the screen, a square area of the screen corresponding to eight by eight character blocks, from any one of six positions (numbered 0-5) to any other of these positions.

Listing 13.3

```
 10 REM CALL &B00 WILL MOVE PICTURE FROM POSITION X% TO POSITION
       Y% (X% AND Y% IN THE RANGE 0-5)
 20 FOR PASS=0 TO 3 STEP 3
 30 P%=&B00
 40 [OPT PASS
 50 LDA HITAB,X : STA &71
 60 LDA LOTAB,X : STA &70
 70 TYA : TAX : LDA HITAB,X : STA &73
 80 LDA LOTAB,X : STA &72
 90 LDX #8
100 .LINLOP
110 TXA : PHA
120 LDY #&7F
130 .BYTELOP
140 LDA (&70),Y
150 STA (&72),Y
160 DEY
170 BPL BYTELOP
180 CLC : LDA &70 : ADC #&80 : STA &70
190 LDA &71 : ADC#2 : STA &71
200 CLC : LDA &72 : ADC #&80 : STA &72
210 LDA &73 : ADC#2 : STA &73
220 PLA : TAX : DEX
230 BNE LINLOP
240 RTS
250 .HITAB : JSR 0 : JSR 0
260 .LOTAB
270 ]
280 NEXT
290 FOR I=0 TO 5
300 READ M
310 HITAB?I=M DIV &100
320 LOTAB?I=M MOD &100
330 NEXT I
339 REM ADDRESS OF TOP LEFT HAND CORNER OF EACH PICTURE (0-5)
340 DATA &5180,&6C00,&6C80,&6D00,&6D80,&6E00
```

We can assemble this routine into memory ready for use by another program. In chapter 10 we gave a program that drew a hidden surface picture of a sphere. Running this program with HORIZ = 12 and VERT = 9 draws an orthographic projection of such a sphere at the centre of the screen. The dimension of the picture just fits in an area equivalent to eight by eight character blocks (position 0). We draw five such pictures of a sphere with 10 vertical segments and 10

SECTION 3 BUSINESS

P20 Loan Repayment Period

This program uses the formula

$$T = \frac{-1}{N}\left[\frac{\log\left(1-\frac{P.R}{N.A}\right)}{\log\left(1+\frac{R}{N}\right)}\right]$$

where

T= period in years.
P= principal.
R= rate of interest.
N= number of payments each year.
A= amount of each repayment.

This could be calculated by using a calculator, but it is far quicker to allow the computer to do the work tor you.

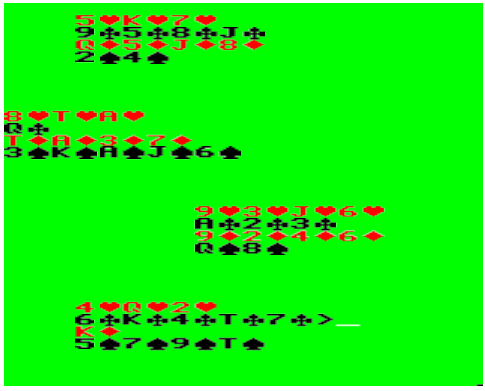
This program could be improved by designing a more robust input routine, to check for bad keyboard input,

COMMANDS

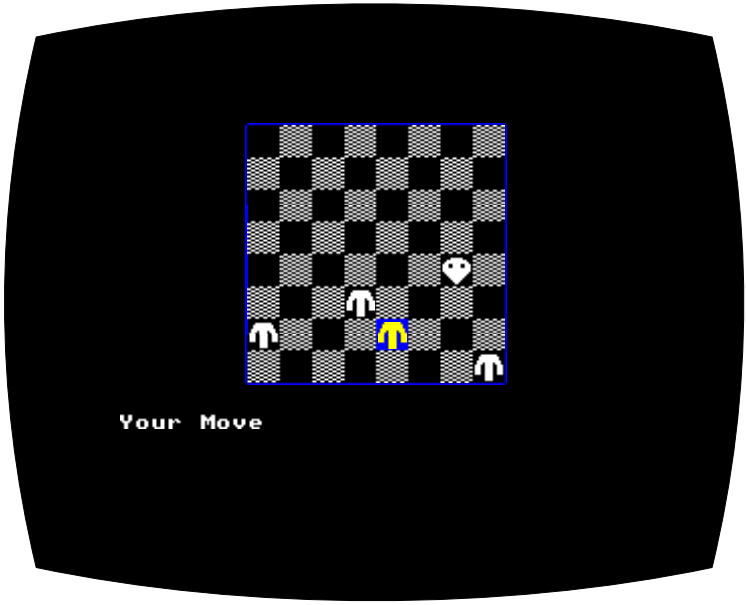
Key in program and type RUN.
Follow instructions.

```
100 REM Program P20 - Loan Repayment Period
110 MODE7
120 PRINT "If you are about to take out a loan "
130 PRINT "it could be useful to consider how long"
140 PRINT "it will be before the loan is repaid."
150 PRINT "To use this program you must input "
160 PRINT TAB(5)"Amount borrowed"
170 PRINT TAB(5)"Annual interest rate"
180 PRINT TAB(5)"Number of payments per year"
190 PRINT TAB(5)"Amount of payments"
200 PRINT '''"Press any key to continue"
210 Z=GET
220 CLS
230 INPUT TAB(5)"Amount borrowed £"P
240 INPUT TAB(5)"Annual interest rate (%) "rate:
    rate=rate/100
250 INPUT TAB(5)"Number of payments per year "N
```

```
  430   hand1$=hand1$+LEFT$(shuffled$,2)
  440   shuffled$=MID$(shuffled$,3)
  450   hand2$=hand2$+LEFT$(shuffled$,2)
  460   shuffled$=MID$(shuffled$,3)
  470   hand3$=hand3$+LEFT$(shuffled$,2)
  480   shuffled$=MID$(shuffled$,3)
  490   hand4$=hand4$+LEFT$(shuffled$,2)
  500   shuffled$=MID$(shuffled$,3)
  510 NEXT I
  520 DIM X(4)
  530 CLS
  540 VDU 19,0,2;0;19,3,0;0;
  550 PROCdeal(hand1$,3,0)
  560 PROCdeal(hand2$,0,8)
  570 PROCdeal(hand3$,8,16)
  580 PROCdeal(hand4$,3,24)
  590 END
  600
  610 DEF FNsetup(suit$,s$)
  620 FOR I=2 TO 9
  630   suit$=suit$+STR$(I)+s$
  640 NEXT I
  650 suit$="A"+s$+suit$+"T"+s$+"J"+s$+"Q"+s$+"K"+s$
  660 =suit$
  670
  680 DEF PROCdeal(hand$,X,R)
  690 X(1)=X:X(2)=X:X(3)=X:X(4)=X
  700 FOR I=1 TO 13
  710   card$=MID$(hand$,I*2-1,2)
  720   IF RIGHT$(card$,1)=h$ OR RIGHT$(card$,1)=d$ THEN COL
OUR 1 ELSE COLOUR 3
  730   Y=ASC(RIGHT$(card$,1))-223
  740   PRINT TAB(X(Y),Y+R)card$;
  750   X(Y)=X(Y)+2
  760 NEXT I
  770 ENDPROC
```



7
Capture the Quark

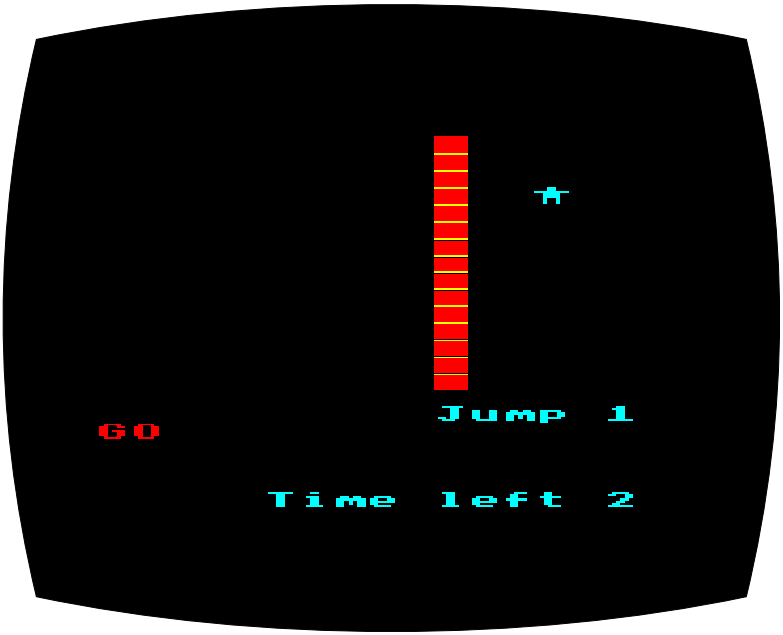


What on earth is a ‘quark’? Well may you ask that question, but to find out you’ll have to play this game. Here are just a few clues. The game is played on an eight-by-eight checkered board and the object of the game is to trap the quark and prevent him from reaching the bottom of the board. To do this you have two, three or four (determined at random) pieces, or ‘quatlins’, which can be moved diagonally one square at a time and only up the board. The quark also moves diagonally but he can move both forwards and backwards.

How to play

At the beginning of the game your quatlins (two, three or four of them according to the luck of the draw) are ranged along the bottom line and the quark is at the top of the board. It is your move first.

8
Commando Jump



This game is a real test of your reaction time and dexterity, and is quite compulsive to play. A bright red wall of varying height appears with a little man figure beside it. A countdown “Ready, Steady, GO” is flashed up on the left of the screen and on the word “GO” the man has to jump as high as possible and then scrabble up the remainder of the wall. Y our success in this game depends entirely on your quick wits and nimble fingers.

How to play

On the word “GO”, and no sooner, press any key to make the man jump. The height of the initial jump depends entirely on the delay between the signal appearing and your key press. The quicker you react, the higher the man will jump. The time left to scale the wall is displayed on the screen and while the rest of your five seconds tick away you must keep on pressing any key to get the man over the

titles and the instructions (when requested from the menu), and provide the data for the program. Great care must be exercised to ensure the data is typed in correctly. The 999 which appears in the data statements after line 1420 acts as a data terminator, but the trailing zeroes are needed to prevent the READ statements in lines 660-700 from failing.

1750-2100 These lines contain the menu procedure, and the text display for the routines to display elements by periodic groupings, with a subsidiary menu for the choice of period. Note the keyboard validations in lines 1910-1960 and 2060-2080.

2110-2800 These lines have the procedure for display of chosen periods of the elements.

2810-3680 Procedures to display particular groups of elements as chosen in lines 3130-3160.

3690-end This sections contains the test sections of the program. The conversion routine in lines 4100-4160 means the program will accept answers both in lower and upper case. The correct answers are displayed when you have finished the test. Both tests are 20 questions at random.

Educational Note

These types of programs are good exercises in information retrieval for upper school youngsters. There is nothing in the program which cannot be found in a book, but the presentation in groupings and periods and the very interaction with the computer has tutorial value. The level of knowledge assumed is good O-level or A-level, and below this standard youngsters will not find it very meaningful. It provides a good quick reference for teachers, but must of course be located before it is needed to fulfil this function.

Program Listing

```
   10 REM ============================
   20 REM   PERIODIC TABLE
   30 REM
   40 REM   BY CW FEB 1983
   50 REM ........................
   60 REM
   70 DIM NAME$(103),SYMBOL$(103),GROUP%(8,8),PERIOD%(7,9),T
RANS%(3,11),RARE%(15),URAN%(15),TEST%(20),ANS$(20)
```

Guess

One of the traditional programs which one can almost certainly find in any book claiming to teach beginners how to program is a program in which the user must try to guess the computer's number, aided by hints such as 'higher' or 'lower'.

When I first became very interested in computers, I spent many hours developing a program which would have the computer guessing the user's number. I finally succeeded and the program in this book incorporates two programs, the first where the user must guess the computer's number as quickly as possible, and the second in which the computer must guess the user's. It is fascinating to watch the computer get closer and closer in guesses to the number hidden in your head, and this program easily fulfils the demands of a program which simulates artificial intelligence — it is impossible to tell whether you are communicating with a computer or a person on a terminal somewhere else. The computer is very good at guessing numbers, and usually guesses my number in less than it takes me to guess its number on average.

HOW IT WORKS

The part where you must guess the computer's number is simple enough. The computer guesses your number by using two variables MIN and MAX and it alters these variables according to the hints {higher or lower) which you provide. A randon element is incorporated to prevent the program developing an endless loop. A count is used to see how many times the computer is calculating a number, and if it gets too high, the computer decides that its guesses are too cold and alters its range accordingly.

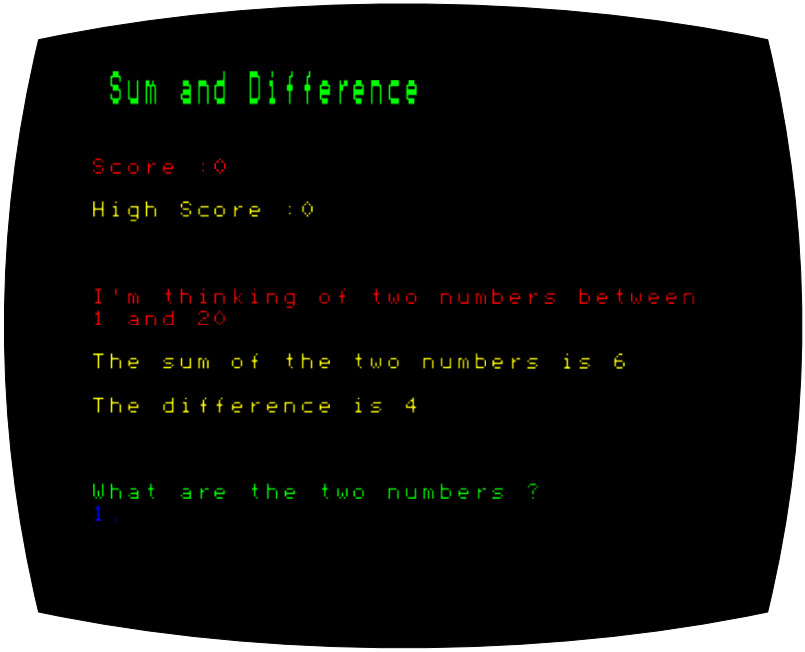
STRUCTURE

Line	Effect
10-100	Main program — calls PROCedures.
PROCIN	Title page
PROCGM	User guesses computer' s number.
PROCGY	Computer guesses user's number.
PROCH	Computer's guess is too high.
PROCL	Computer's guess is too low.
PROCS	Computer's guess is fine-tuned.
PROCR	Random element
PROCW	Displays winner.
PROCHI	Prints 'higher'.
PROCLO	Prints 'lower'.
PROCO	Prints 'correct'.
PROCT	Second title page.
PROCGL	Computer is jubilant.

LENGTH IN BYTES
2164

RUNNING ON A MODEL A
No modifications required.

5
Sum Difference



We thought about calling this game 'some difference' as it isn't nearly as easy as it looks at first.

You are being told by your computer that it is thinking of two numbers between 1 and 20. It will tell you the total sum and the difference between the two numbers. All you have to do is correctly guess the answer.

Example: The sum of the numbers is 13
The difference is 9
What are the numbers?
Answer 2 and 11

Simple isn't it?

How to play

Your computer will tell you the sum of the numbers it is thinking of and the difference and ask for your answers.

After each number press RETURN.

If you are correct the score increases on the top of the board.

As you become better at this game the computer will move the range of numbers from 1-20 to 1-25 and so on.

A wrong answer will end the game completely and you will be asked if you wish to compete again. High scores will be recorded on the screen to allow you to compete for the high score title.

Programming Hints

You can make the game tougher from the very beginning by increasing the value in line 20 so that the range is wider immediately.

Program

```
   10 REM Sum and Difference
   20 High=0:Score=0:Maximum=20
   30 MODE7
   40 REPEAT:CLS
   50 PROCHeader
   60 PROCScore
   70 Rand1=RND(Maximum)
   80 REPEAT
   90 Rand2=RND(Maximum)
  100 UNTILRand2<=Rand1
  110 PRINTTAB(0,11);CHR$129;"I'm thinking
 of two numbers between"'CHR$129;"1 and ";
Maximum
  120 PRINTTAB(0,14);CHR$131;"The sum of t
he two numbers is ";Rand1+Rand2
  130 PRINTTAB(0,16);CHR$131;"The differen
ce is ";Rand1-Rand2
```

Division One (Model A)

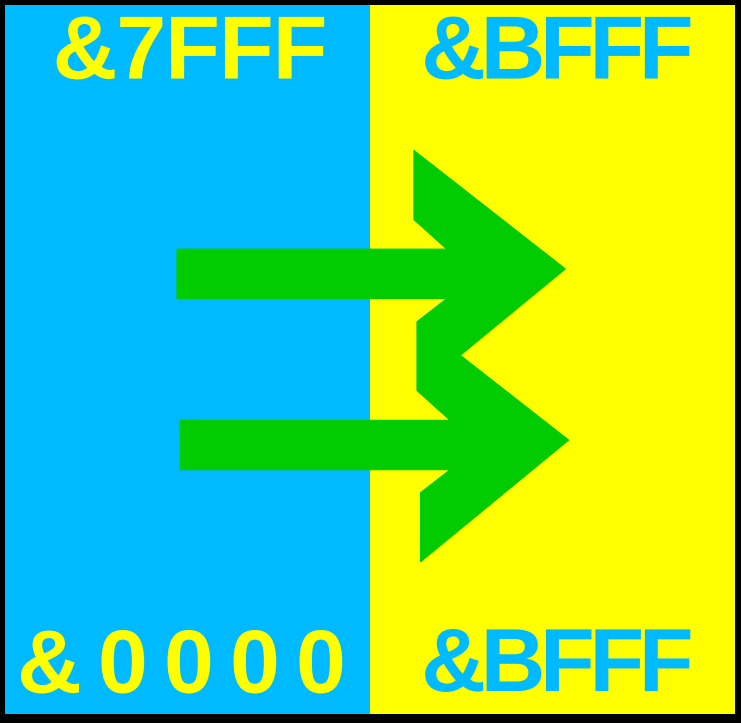
This time around you're a soccer manager whose overall proficiency will be judged at the end of a season. There are details of matches played and goals scored, points awarded, games remaining and a constantly changing league table. The computer will calculate the results according to the relative strengths of the teams, whether the game was played at home or away, and so on.

If things are going badly it could be that you'll have to intervene at the touchline, changing players' positions and generally backseat booting. Is it going to be championship or relegation? It's up to you and the lads. And the DATA statements which can be messed around with if you feel that we've been biased one way or another.

```
   10 REM BBC VERSION ************
   20 REM DIVISION ONE.....WALWYN
   30 ON EROR GOTO 2660
   40 P5=0 : PL=0
   50 MODE 7 : VDU 14,23,1,0;0;0;0;
   60 PROCinstructions
   70 J=0
   80 DIM TZ(15,15),T$(15),TA(15),TM(15),TD(1
5),TT(15),TP(15),TF(15)
   90 DIM D(1,9),XS(1,6),YS(1,6),X(1,6),Y(1,6
),S(1,6)
  100 FOR I=1 TO 15 : READ T$(I),TA(I),TM(I),
TD(I) : NEXT I
  110 DATA "LIVERPOOL",9,6,6,"MAN UTD",8,5,6,
"IPSWICH",6,7,6,"ARSENAL",7,6,6
  120 DATA "STHMPTON",8,5,5,"A VILLA",6,6,5,"
NOTTM F",9,5,5,"SWANSEA",5,6,5
  130 DATA "WOLVES",5,5,5,"CRYSTAL P",5,6,5,"
TOTTENHAM",8,4,6,"NORWICH",4,5,4
  140 DATA "COVENTRY",4,4,4,"LEEDS",4,3,4,"W
BROM",8,3,4
  150 FOR X=0 TO 1 : FOR Y=0 TO 9 : D(X,Y)=Y+
128-X*128 : NEXT Y : NEXT X
  160 FOR J=1 TO 6 : READ XS(0,J),YS(0,J) : N
EXT
  170 FOR J=1 TO 6 : READ XS(1,J),YS(1,J) : N
```

THE ADVANCED BASIC ROM USER GUIDE

FOR THE BBC MICRO



Published by the Cambridge Microcomputer Centre

COLIN PHARO

9 TRIGNOMETRICAL MANIPULATIONS

The previous chapter gave typical timings for all of the BASIC subroutines. Inspection of these timings reveals the fact that trignometrical functions are especially time consuming. There are a number of different methods which can often be used to get round this problem and this chapter explains many of therm Each method is illustrated by the polygonal circle discussed in Chapter 5, but the methods are applicable to many situations in which trignometry is used. It should be remembered that the conventional method of drawing circles takes nearly 6 seconds in BASIC and even in machine code requires 5.5 seconds. With a little chicanery, considerable improvements on these times may be achieved. Apart from the first method which must use assembly language, BASIC is used in demonstration programs so that the methods are easier to understand.

9.1 Fixed Shapes Method

The following program illustrates a method that can be used whenever the application draws a geometric shape of fixed dimensions in a fixed position. In this method, all the coordinates to be plotted are stored as constants within the program. In the demonstration program, the two functions, XCOORD and YCOORD respectively, store away all 100 X and Y coordinates of the circle to be plotted. The program itself simply plots these points. All of the BASIC parts of the program arc disposable. The generated machine code draws the circle in 0.28 seconds. Of course, this method uses a lot of memory to store coordinates (404 bytes in this example). Moreover, it is not a general purpose routine to draw many circles of different sizes. Nevertheless. it is the quickest method and is useful in many applications.

Exercise 6.2
Write variations on this standard 'histo' procedure that can be substituted into the complete package as and when required. For example write a procedure that draws the histogram as a set of pairs of bars. The space between any two bars that form a pair should be half the distance between neighbouring bars that do not form a pair. Use this to construct diagrams that are similar to figure 6.2.

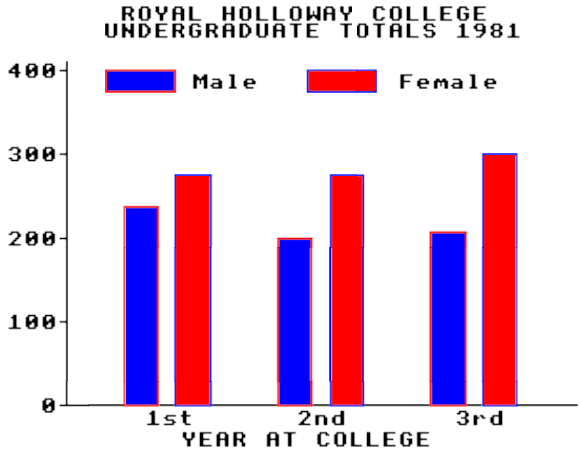


Figure 6.2

Example 6.2
In listing 6.3 we give an example of such a replacement 'histo' procedure. This version of 'histo' (using a variation on the fake-perspective cube procedure from chapter 1) produces an apparently three-dimensional graph. Two data values are requested for each bar, a MAXimum and a MINimum; the maximum bar is drawn behind the minimum bar. This program can be used to create charts similar to figure 6.3 which shows the monthly temperature variation in Egham.

Exercise 6.3
There are many, many more possible variations, for example drawing bars above and below a central line in order to display fluctuations in currency exchange rates. See the Money Programme on BBC2 for ideas. The fundamental notions we have introduced here should enable you to produce histograms to your own specifications.

assume that the plane of symmetry is the *y*/*z* plane, and so for every point (*x*, *y*, *z*) on the jet there is also a corresponding point (–*x*, *y*, *z*). To draw figure 9.3 we use all the graphics and 4 × 4 matrix routines, listing 9.1 and 9.2, together with listing 9.9, 'scene3' and construction procedure 'jet' which generates all the vertices of the aeroplane that have positive *x*-coordinates, and thus stores information only about one-half of the jet. To construct the complete aeroplane we also need a 'drawit' procedure (also in listing 9.9) which draws one side of the jet and then, by reversing the signs of all the *x*-values, draws the other.

It is simple to construct these figures, just plan your object in various sections on a piece of graph paper, number the important vertices and note which pairs of vertices are joined by lines. The coordinate values can be read directly from the grid on the paper. The data for figure 9.3 are HORIZ = 160, VERT = 120, (EX, EY, EZ) = (1, 2, 3) and (DX, DY, DZ) = (0, 0, 0).

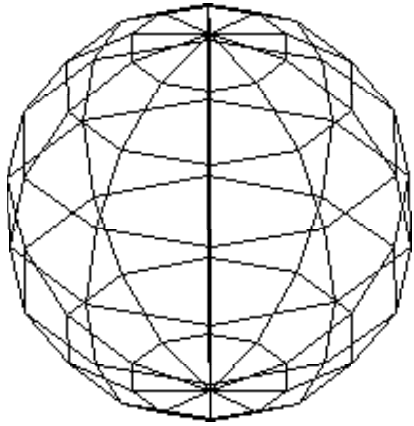


Figure 9.4

Bodies of Revolution

This far in our construction of objects we have relied on DATA to input all the information about lines and vertices. We now consider a type of object where only a small amount of information is required for a quite complex object – this is a body of revolution, an example of which is shown in figure 9.4.

The method is simply to create a defining sequence of NUMV lines in the *x*/*y* plane through the origin; this is called the definition set. We then revolve

two-dimensional string arrays. It will also include routines which sort fixed length multi-field records. They are given complete with BASIC parameter passing lines so they can be entered and exhaustively tested. It should be pointed out that the machine code portion of the listings will stand alone as subroutines, providing that:

(a) the correct parameters are passed from any BASIC program;
(b) the code is lodged either in one of the safe areas (not necessarily the areas used in our listings) or dynamically, above or below BASIC, by the use of the DIM statement.

All the programs in this chapter are concerned with sorting data into either numerical or alphabetical order and will be useful in compiling indexes, customer lists, domestic accounts, hobby collections (butterflies, stamps, beetles) etc. They could also be employed as routines within general purpose filing systems to store or retrieve information according to some predetermined order.

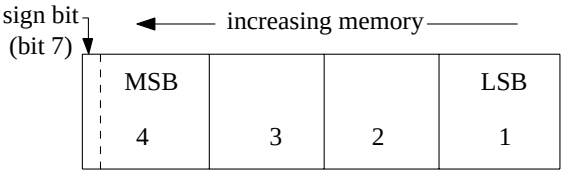
Bubble sort of a BASIC integer array

The bubble sort is well-known but often despised because it is slow. It is one of the simplest sort routines to understand and, providing there are not too many elements in the array, can be acceptable if written in machine code. It provides a good starting point for handling multibyte integers.

Because the programs which follow are intended to be used in conjunction with BASIC, via CALL parameters, it is important to understand how the interpreter allocates variable space.

How integer array variables are stored

The four bytes, allocated to each integer array variable, are arranged as follows:



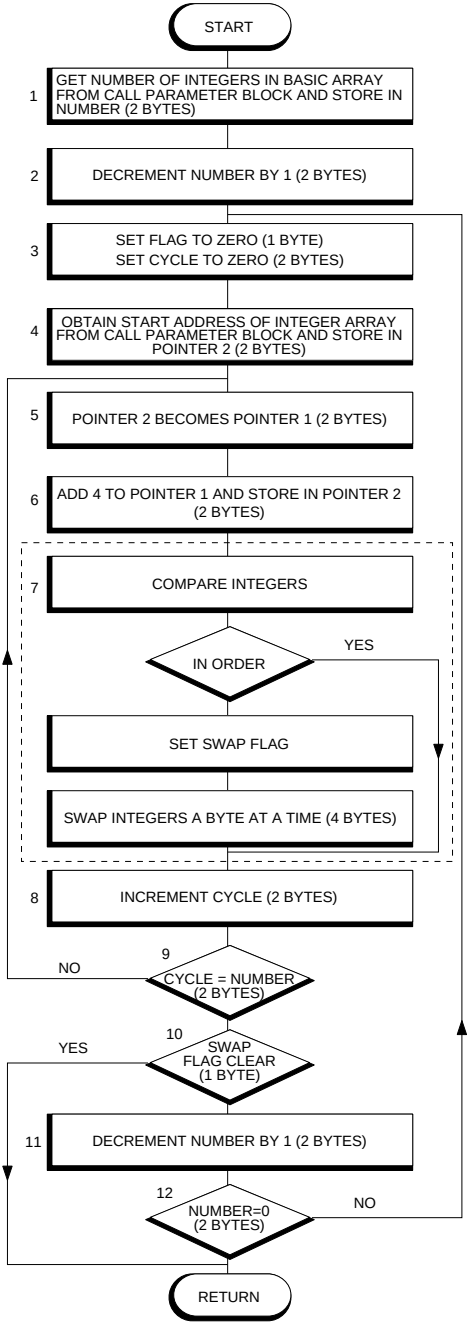


Fig. 6.1. Flowchart for integer array bubble sort.

to store DATA in the Ith element. Two-dimensional arrays are just as easy; the only real change is that the expression in lines 1020 and 20l0 becomes:

```
PTR# F=I*6+N*J*6
```

to access the I,Jth element of the array.

Using files from assembler

The BBC Micro's filing system is almost as easy to use from 6502 assembler as from BASIC. There is a great similarity between the MOS routines used to manipulate files and the equivalent BASIC commands. For example, the routines OSBPUT and OSBGET will write and read a single byte in the same way that BPUT and BGET do. Table 6.1 gives details of the MOS routines corresponding to each of the BASIC file operations:

Table 6.1

BASIC	MOS routine	parameters
OPENIN	OSFIND (&FFCE)	A=&40, Y,X address of file name. On return Y contains the channel number. Y=0 if OSFIND cannot open the file
OPENOUT	OSFIND (&FFCE)	As for OPENIN but A=&80.
OPENUP	OSFIND (&FFCE)	As for OPENIN but A=&C0.
CLOSE	OSFIND (&FFCE)	A=0, Y=channel number of file to be closed. (If Y=0 then all files are closed.)
BPUT	OSBPUT (&FFD4)	A=byte to be written, Y=channel number
BGET	OSBGET (&FFD7)	Y=channel number. On return A contains byte read from file. Carry flag=1 if an error has occurred in which case A contains an error code. (&FF is 'end of file')

There are other MOS routines concerned with file handling but the ones given in Table 6.1 are those most often used. For example, there is a MOS routine, OSFILE, that performs the same action as the BASIC commands SAVE and LOAD. There are also a number of routines that do not apply to files stored on cassette. In particular, the needs of random access disk files are catered for by OSARGS (&FFDA). On calling this routine the X register should contain the address of the start of four memory locations in page zero. These are used to hold the input value, or the result of calling OSARGS, in the usual four byte integer format. Calling OSARGS with a channel number in Y will read the file's current position pointer if A=0, write the current position pointer if A=1, and read the file's length if A=2. A call to OSARGS with A = &FF will ensure that any alterations made to the file are actually written out rather than just sitting in a buffer.

OSARGS also has a number of other functions including returning the code of the currently active filing system. This was used in the function FN file system that can be used from BASIC to discover which type of file device is in use. There is nothing complicated about using these filing system routines as they provide the same set of operations as their equivalents in BASIC.

A disk sector editor

One of the most useful utilities that any disk user can possess is a *sector editor*. A sector editor will read in any sector of a disk and display it in hex or in ASCII characters and then allow you to write it back to disk after making any changes to the data that are necessary. This may sound like a difficult program but the disk filing system includes an extension to the MOS routine OSWORD to read or write a sector. Calling OSWORD with the A register set to &7F will read or write a sector according to the contents of a parameter block.

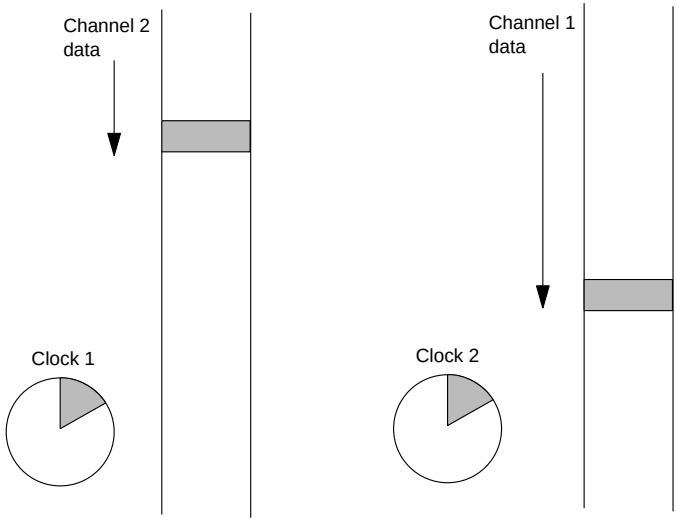
byte	*meaning*
0	drive number
1-4	address of sector buffer
5	3
6	&53=read sector &4B=write sector
7	track number
8	sector number
9	&21

This has to be set up before entering OSWORD and its address stored in the X and Y registers.

Using this information a sector editor is easy to write:

```
  10 REM SECTOR EDITOR
  20 DIM SEC_BUF% 255
  30 DIM INS_BLK% 50
  40 MODE 4
  60 PROCparm_get
  80 PROCsect_op
  90 PROCsect_print
 100 GOTO 60

1000 DEF PROCparm_get
1010 PRINT TAB(0,28);
```



In general the current note for each channel will be in a different position in the data streams. The clocks will tend to show equal elapsed times. Each time a SOUND statement is obeyed, the duration of the note is added to the clock associated with that channel. At each step we must obey a SOUND statement for the channel whose clock shows the least elapsed time. We require to repeat the following operation:

```
IF clock1 > clock2  THEN SOUND statement for channel 2

                    ELSE SOUND statement for channel 1
```

The program then selects one out of two alternative courses of action and this ensures that the channels free run and are not subject to interference from each other. Effectively we have removed the artificial connection in the parallel data streams between notes in different channels that have different duration values.

An alternative method of playing notes of a two-voice melody from separate data streams is to use the function ADVAL to test the channel queue status. For example, the expression 'ADVAL(-6)' has a value indicating the number of empty places on the channel 1 queue (-7 for channel 2 and -8 for channel 3). Thus, we could use:

```
IF ADVAL(-6)>0 AND voice 1 not yet finished THEN
                          SOUND statement for channel 1
IF ADVAL(-7)>0 AND voice 2 not yet finished THEN
                          SOUND statement for channel 2
```

This ensures that no SOUND statement is obeyed if a channel queue is full. The end effect is exactly the same as that of the clock algorithm. Using ADVAL, a separate test is needed In check whether a voice has finished, whereas with the clock method this possibility can be dealt with by setting the clock to a large value when the last SOUND statement for that voice is obeyed. The clock algorithm also makes it easy In incorporate a common musical requirement - emphasis of tho first note in every bar. Here the state of a clock could be used to recognise the first note of a bar.

Transposing

Another tedious task to be overcome before we start getting the machine to play arrangements is transposing from a musical score to a set of pitch numbers and associated notation. Transposing directly from the black dots to pitch numbers and durations in 1/20ths of a second can be tedious and error prone. You can write a graphics 'picking and dragging' program to input the music onto a screen stave, and this is a commonly adopted approach, but we have not space for that. Instead we shall adopt a character convention, and list the music in DATA statements using the following tables.

code	musical convention	duration (for metronome 150)
t	1/32	1
s	1/16	2
ds	dotted 1/16	3
e	1/8	4
de	dotted 1/8	6
q	1/4	8
dq	dotted 1/4	12
h	1/2	16
dh	dotted 1/2	24
w	whole	32

Remember that there are notes that cannot be accurately represented at this tempo. For example a dotted 1/32 is 1.5 (only 1 or 2 can be used as a duration parameter in a SOUND statement). Similarly a 1/16 triplet is 4/3 per note, an 1/8 triplet 8/3 per note and a 1/4 triplet 16/3 per note.

Chapter 4 **Animation techniques**

The commonest animation technique that you are likely to use on your Electron will be character animation, where objects are moved about the screen by printing and reprinting characters. In any of the graphics modes, a character shape can be displayed by a PRINT statement in considerably less time than it would take to draw the same shape using graphics commands. This is because of the fast techniques used to fill the area of the screen memory that is to be occupied by the character. In MODE 7, character printing is even faster than in the graphics modes.

We shall see later in the chapter how to define our own character shapes but for the time being, the objects being moved will be strings of standard characters. Such simple animation of words and numbers is a powerful tool in computer-assisted learning systems as we shall demonstrate shortly.

Although the use of DRAW and PLOT facilities for nnimation is limited by lack of speed, towards the end of the chapter we shall look into techniques for the animation of simple line drawings such as stick figures.

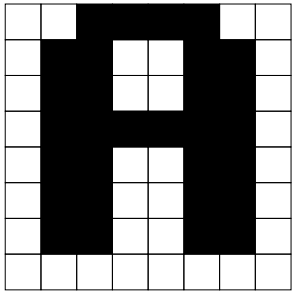
4.1 User-defined characters

In modes 0, 1, 2, 4 and 5, the screen is divided up into a number of 'pixels'. For example, in modes 1 and 4, there are 320x256 pixels.

In modes 3 and 6, the screen is divided into horizontal strips of pixels which are separated by strips of background colour. Each strip is 8 pixels deep.

In any of modes 0 to 6, printing a character has the effect of filling an 8x8 group of pixels with a pattern of foreground and background colour. For example, the pattern for "A" shown on the next page.

Also associated with each character is an ASCII code number in the range O to 255. This code is used inside the computer to refer to the character. The ASCII code for "A" is 65. When the character whose code number is 65 is to be displayed on the screen by a PRINT statement, the above pattern of foreground and background colour is inserted into the screen memory where information is stored about what is currently displayed on the screen.



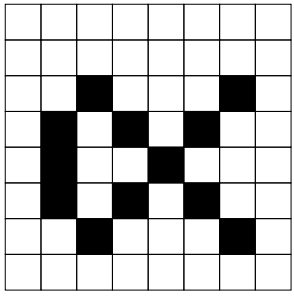
The user is normally free to define the character shapes that are associated with ASCII code numbers 224 to 255, and this is particularly useful when creating shapes for use in animation. In fact, it is possible for the user to define shapes for a much greater range of ASCII codes. We shall explain how to do this shortly.

Once a new character shape has been defined, it can be displayed on the screen at the same speed as the predefined characters that we have used so far in this chapter.

The use of user-defined character shapes has two advantages over the use of PLOT instructions to draw shapes. Firstly, as we have already seen, a character shape is dislayed on the screen at a much greater speed than can be achieved by using PLOT facilities. Secondly, the sequence of PLOT statements needed to draw a complex shape such as a spaceship would be rather lengthy.

Single character shapes

Let us demonstrate the process of defining new character shapes by defining some Greek letters. These could be useful to a scientist or a mathematician wishing to display muthematical equations. The shape required for alpha is



Note that in MODES 2 and 5, a pixel (and therefore a character) is elongated horizontally. Each row in the 8x8 pattern can be viewed as a byte (eight bits - see Appendix

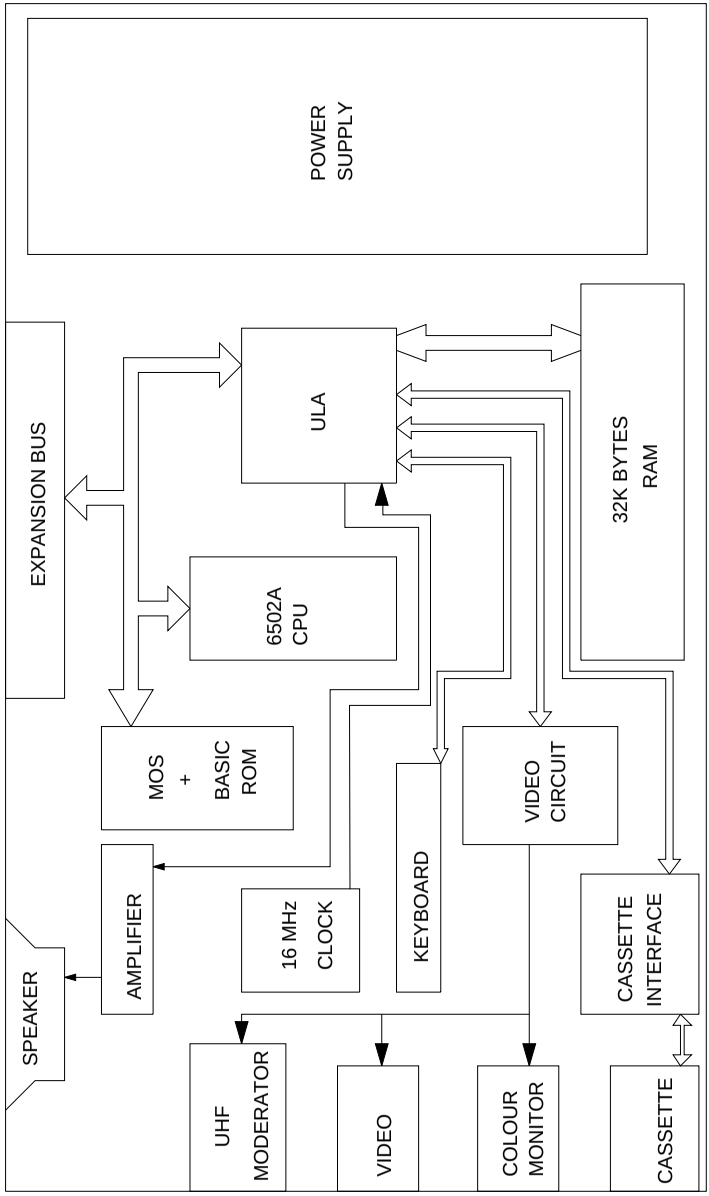


Figure 13.1 The system block diagram

running together at the same rate. The RESET line allows all hardware to be initialised to some predefined state after a reset. An interrupt is a signal sent from a peripheral to the 6502 requesting the 6502 to look at that peripheral. Two forms of interrupt are provided. One of these is the interrupt request (IRO) which the 6502 can ignore under software control. The other in the non-maskable interrupt (NMI) which can never be ignored. Refer to chapter 7 on interrupts for more information.

When power is first applied to the system, a reset is generated by the ULA to ensure that all devices start up in their reset states. The 6502 then starts to get instructions from the ROM. These instructions tell the 6502 what it should do next. A variety of different instructions exist on the 6502. The basic functions available are reading or writing data to memory or an input! output device and performing arithmetic and logical operations on the data. Once the MOS (machine operating system) program is entered, this piece of software gains full control of the system.

On an unexpanded Electron, the computer will continue operating under the MOS until it is switched off. Programs are entered into the memory from the keyboard or cassette port, then run. There is some scope for clever programming techniques using the standard hardware - they all involve some tampering with the various registers in the ULA, However, a lot more facilities can be provided by adding extra hardware onto the back of the Electron.

Since the Electron is the little brother of the BBC Micro, two forms of expansion are provided for. The first of these covers the addition of hardware which is supplied as standard on a BBC Micro. Within this category are included items like a printer port, analogue to digital converter (for joysticks) and paged ROMs. The second category includes items which would have to be added onto a BBC Micro. Products like the second processors and units which plug onto the One Megahertz Bus are in this category.

10 ROM Routines

Many of the tasks which need to be performed when dealing with the BASIC system are handled by standard routines inside the BASIC ROM. There are standard routines for expression evaluation, checking the syntax of lines, handling the memory allocation, and arithmetic routines. Although some of these will only be of use inside new statements and functions (like the ‘Get character at PTRB’ routine); many can be used from simple machine code programs, to allow floating point calculations to be performed, or accessing the variables passed by the BASIC ‘CALL’ statement, perhaps.

Note that these ROM routines can only be used if BASIC is paged in to &8000 to &BFFF. If the machine code program which uses them will be called from BASIC, using either the ‘CALL’ statement or the ‘USR’ function, BASIC will be paged in. The programs in chapters 7 to 9 rely on this. However, BASIC will not be paged in if the program is called by using the ‘*RUN’ command in any filing system which itself sits in a paged ROM (like DFS, for example): the filing system ROM will be paged in instead.

To check that the current paged-in ROM is BASIC, the RAM copy of the paged ROM select register (in location &F4) should be compared with the ROM number of the BASIC ROM. This can be found by using OSBYTE &BB (187). For example, this section of code will check that the current ROM is BASIC:

```
LDA #&BB          \Call OSBYTE &BB to read the ROM
LDY #&FF          \ socket number containing BASIC.
LDX #&00          \ X and Y are set to read it without
JSR osbyte        \ modification.
CPX &F4           \If it is not the same as the current
BNE giveup        \ ROM, don't continue.
```

The BASIC ROM does not need to be paged in if the only part of the machine code program which is to be ‘RUN’ is the initialisation section, and that just needs to check the year of the BASIC ROM (but uses no ROM routines). If this is the case, the BASIC ROM slot number can be found using OSBYTE &BB as

above, and the year byte found by using OSRDRM (&FFB9). For example, the following code will read the year byte of the BASIC ROM:

```
LDA #&BB          \Call OSBYTE &BB to read the ROM
LDY #&FF          \ socket number containing BASIC.
LDX #&00          \ X and Y are set to read it without
JSR osbyte        \ modification.
TXA               \
TAY               \Transfer the ROM number into Y,
LDA #&80          \ and call OSRDRM to read the byte
STA &E7           \ at location &8015 in the BASIC ROM
LDA #&15          \
STA &E6           \
JSR &FFB9         \
```

Note that OSRDRM was implemented for operating the '*ROM' filing system in paged ROMs, so use it with caution (as with most of the rest of the examples in this book!).

Throughout this section, I have used the names of many of the standard BASIC registers, rather than the actual memory they occupy. They are detailed in other parts of this book, but here is a summary of them:

IntA This is the integer accumulator which is held in page zero at &2A to &2D (LSB in &2A, MSB in &2D). It is used in integer calculations, and also to pass integer values between routines.

The low 3 bytes of IntA (&2A to &2C) are also used to hold the variable descriptor block when handling variables. When being used for this, &2A and &2B point to the first byte of the variable value, and &2C contains the variable type (for a description of the variable types, see section 3.1.3). This variable desrcriptor block is sometimes used at &37 to &39 (if IntA is used to hold the value of the variable).

FPA This is the main floating point accumulator, which is held in page zero at &2E to &35 (see section 2.2.2 for the floating point accumulator format). It is used in calculations involving real numbers (together with FPB), and also to pass real values between routines.

PATIENCE

BENNY NOTARIANNI
Original program by Richard Still

GENERAL DESCRIPTION

The user plays a game of Patience on the screen. The object is to place each suit in ascending order on its respective pile.

Instructions are displayed, then the main program is called (It is not possible to look at the instructions again without rewinding the tape?) The single pack is laid out in the form of a 'Harp'. The cards may be moved from one column to another or new cards. are dealt from the stack As soon as an Ace becomes available, it may be used to start a file.

No false moves or cheating are allowed! Perhaps a 'Cheat' facility could be added for desperate players?

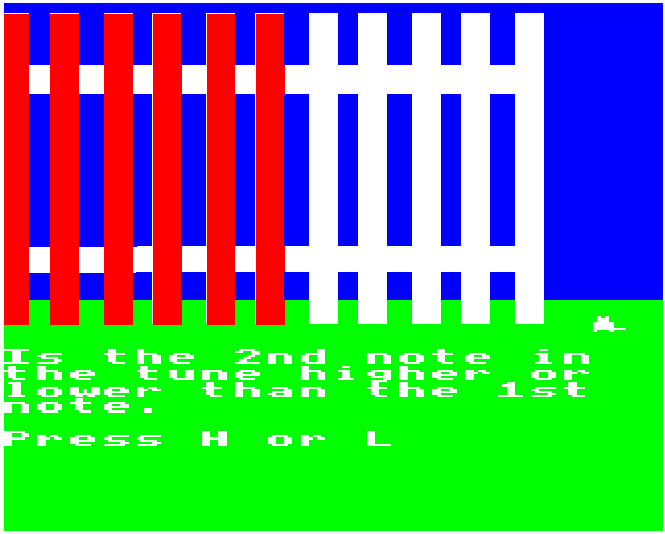
DETAILED DESCRIPTION

Program 1 Introduction
Lines 10-190 the introductory program
170 resets PAGE to allow extra memory for disc users.
180 calls the actual game program
210-420 instructions
440-560 defines the pips and the back of the cards. (Thus saving space in Program 2.)

```
  1 REM *INSTRUCTIONS/INITIALIZE PROGR
AM
 10 MODE7
 20 PROC_pchars
 30 PRINT CHR$141;CHR$130;SPC(10);"PAT
IENCE"
 40 PRINT CHR$141;CHR$131;SPC(10);"PAT
IENCE"
 50 PRINTSPC(14);CHR$134;"BY"
 60 PRINTSPC(7);CHR$133;CHR$136;"B. NO
TARIANNI"
```

```
   70 A=INKEY(300)
   80 VDU 30
   90 FOR X%=1 TO 9
  100   VDU 11
  110   A=INKEY(25)
  120 NEXT
  130 VDU 31,3,15
  140 PRINT" Do You want Instructions."
  150 *FX15,1
  160 IF LEFT$(GET$,1)="Y" PROC_instruct
ions
  170 IF PAGE=&1900 THEN PAGE=&1100 : RE
M DISK USERS ONLY
  180 CHAIN"CARDS"
  190 END
  200
  210 DEF PROC_instructions
  220 CLS
  230 PRINT" This is a computerised vers
ion of"
  240 PRINT"the game of Patience."
  250 PRINT" Cards are dealt to a stack
3 at a"
  260 PRINT"time by pressing D. The play
er may"
  270 PRINT"move a card from the stack t
o the"
  280 PRINT"columns by specifying S as t
he"
  290 PRINT"source and the column number
 as"
  300 PRINT"destination."
  310 PRINT" Aces can be removed from th
e stack"
  320 PRINT"or columns and placed in the
ir own"
  330 PRINT"boxes by specifying P as the
 TO"
  340 PRINT"parameter."
  350 PRINT" The game is won by placing
all"
```

DON'T PAINT THE CAT



Seems a strange title for a program. I mean, who would want to emulsion paint the family mogg anyway?

Well you see, the family have decided that you have to paint the garden fence. You lost the draw - it might have been your sister instead who had to do it, but never mind there is always next time. Across the fence from you and your fantastic paint brush, is your neighbour's transistor. As a mental challenge you have decided to paint the fence according to the high/low pitch of your neighbour's music.

Look out for your cat, it's parked at the end of the fence.

How to play

As the game begins you will hear just two notes to compare but, everytime you get the answer correct the next tune will have an extra note.

You will be told which two notes to compare, and you must key in H or L for High or Low.

If you get it wrong you must wait for the fence, and the poor old pussy, to be painted.

If you take too long to answer, the cat will wind up getting covered in paint anyway.

Press the RETURN key when you want a new tune.

Programming hints

If you can work out the answer long before the cat is painted, then reduce the 50 if INKEY$(50) in line 540.

If you find that it is too difficult to tell the difference between notes, then increase the 5 after the '*' sign in line 470.

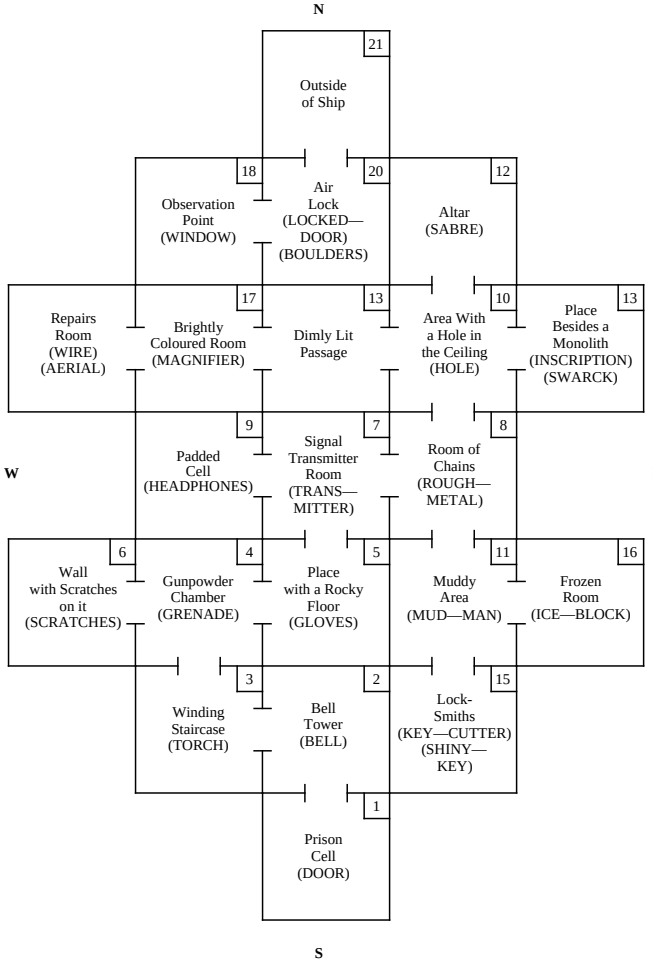
Alternateively you can increase the time allowed to answer, or reduce the difference between the notes, by doing the opposite of what is described above.

```
  10 REM DON'T PAINT THE CAT!
  20 REM COPYRIGHT (C) G.LUDINSKI 1983
  30 MODE 5
  40 DIM N(10)
  50 CLS
  60 VDU 19,0,4,0,0,0,19,2,2,0,0,0
  70 GOTO 170
  80 REM
  90 REM U.D.G. CALCULATOR
 100 REM
 110 DEF FNB(N$)
```

CHAPTER 2
A MODEL ADVENTURER

The Plan

The first stage in writing an adventure is the making of a plan with the layout and labelling of the rooms with a system of numerical values for the objects in each room along with the room number and name:



As can be seen from the diagram, the names of the rooms are given along with the room number for each room, and the objects associated with the individual rooms are placed in brackets inside the room square.

Although the method for numbering rooms may at first appear not to be logical, the rooms are in fact in a logical order. Room “I” is where the player starts from, and room “2” is the first room that can be entered from this room. Rooms “5” and “6” are to the right and left of room “4” — I give preference in numbering to north then south then east and then west. I build up the numbering system by looking for the room with the smallest numerical value which has an exit, or exits, to an unnumbered room, or rooms (numbering is done in the above priority). The above method works because rooms beside each other have the lowest possible difference between their numerical values — this is a key feature, especially in larger adventures, in the movement between rooms, which is dealt with later in this chapter.

If this seems a little confusing, then the step by step process for deciding the numerical values for the rooms in the given example is as follows: the only exit from room “1” is room “2”, and the only unnumbered room from room “2” is room “3”. From room “3”, the only yet unnumbered room, is room “4”. However, from room “4” there are two unnumbered rooms, the eastern being numbered first as room “5”, and the western second as room “6”. At this point, the lowest numbered room is room “5”, and the only room off this room without a number is labelled room “7”. Room “6” is a dead end, so no further rooms may be numbered from this room. Room “7” is another room which has two unnumbered rooms off it, the eastern being labelled room “8”, and the western, which is a dead end, is labelled room “9”. From room eight, there are two rooms yet to be numbered — the northern is designated room “l6”, and the southern, room “11”. The room with the lowest numerical value here, with other rooms off it wanting numbers, is room “1”, and rooms “12”, “13”, and “14” are to the north, east, and west of it — rooms “12” and “13” are both dead ends. After dealing with room “16”, the next room to be dealt with is room “11” : the rooms south and east of this room are two more dead ends, and are numbered with “15” and “l6” respectively. After this fairly complex branching, the only room left that is not a dead end, with a yet unnumbered room off it, is room “14”, and the room off this, is given the next number, which is “17”. Two rooms requiring numbers are north and west of room “I 7”, and are labelled with “18” and “19”, the latter being a dead end. Off room “18” is room “2fl” to the east, and off room “2fj” to the north is the final room, room “21”.

WORKING OUT OF THE DISPLAY

1) Display of the Room Name — Type in the following lines into your computer (if you own a computer other than the BBC Micro, then see Chapter Four for conversion notes):

DIGITALLY REMASTERED EDITION

MIKE JAMES

CREATIVE ANIMATION AND GRAPHICS ON THE BBC MICRO

The reason for the apparently redundant final 1 is difficult to explain in detail but, put in simple terms, it is necessary to allow translations to be included in the matrix formulation. (In fact the use of three numbers to represent a point in two dimensions is a very useful mathematical technique called *homogeneous co-ordinates*.)

The result of a transformation is another column vector obtained by multiplying the original column vector by a 3 × 3 matrix. The multiplication is such that the first element of the new column vector is obtained by multiplying each element of the first row of the matrix with the corresponding element of the column vector and adding the results together. The second element of the new column vector is obtained by performing the same operation using the second row of the matrix and the third element is obtained using the third row. The operation of forming each element of the new column vector is often thought of as multiplying each row of the matrix by the column vector (see Fig. 8.1).

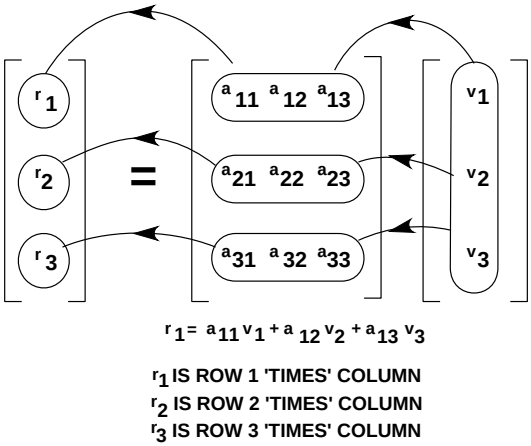


Fig 8 1. The method of matrix multiplication.

As a BASIC program this gives

```
FOR I=l TO 3
FOR J=1 TO 3
A(I)=T(I,J)*V(J)
NEXT J
NEXT I
```

assuming that the initial co-ordinates are stored in V, the result in A and that the transformation matrix is T.

To convert a pair of co-ordinates into a column vector all that you have to do is add the final 1. However, converting a column vector back to a pair of co-ordinates is not always as simple as ignoring the final 1. The reason for this is that during matrix multiplication it is possible for the final I to be changed into some other value. In other words the general form of a column vector is

$$\begin{vmatrix} x \\ y \\ w \end{vmatrix}$$

where w might not be 1. To convert this general form of column vector into a pair of co-ordinates it is necessary first to divide each element by w, giving

$$\begin{vmatrix} x/w \\ y/w \\ 1 \end{vmatrix}$$

The final pair of co-ordinates are then simply x/w and y/w. Transformations that change the value of the final 1 are very important in three-dimensional graphics, but in two-dimensional graphics they can be avoided and the co-ordinates can be recovered by simply throwing away the last element of the column vector.

Some transformations

So far all this talk of matrices is a little abstract and a few examples are long overdue. The matrix R

$$\begin{vmatrix} \text{COS A} & \text{-SIN A} & 0 \\ \text{SIN A} & \text{COS A} & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

produces an anti-clockwise rotation about the origin through an angle A. That is, the point given by RV is the same distance from the origin as the point V but moved through an angle A anti-clockwise. If every point in the point file is multiplied by this matrix and then the lines in the line file redrawn the resulting shape will be the same but rotated through an angle A about the origin. Notice that to produce a rotation

DIGITALLY REMASTERED EDITION

CREATIVE ASSEMBLER

How To Write Arcade Games

for the BBC Microcomputer Model B
and Acorn Electron

Jonathan Griffiths

PENGUIN ACORN COMPUTER LIBRARY

5

ADDRESSING MODES

So far we have met two addressing modes. One of these is absolute addressing as in

```
LDA addr
```

which, when executed, loads the accumulator with the contents of the location whose address is 'addr'. The other is immediate addressing as in

```
LDA #&81
```

which, when executed, loads the accumulator with the actual value &81.

However, other addressing modes exist and one of the most important, 'indexed addressing', is introduced here prior to a summary exposition of all the addressing modes available to the 6502 processor.

5.1 Indexed addressing

In this addressing mode one of the index registers (X or Y) is added to the address as an offset which gives the precise location for the stored data. For example, we can write:

```
LDA addr, X
```

If X contains zero this instruction will behave just like 'LDA addr'. However, if X contains 1 it will load the accumulator with the contents of 'one location further on from addr'. Since X can contain any value from 0 to 255, the instruction 'LDA addr,X' gives you access to 256 different memory locations. If you are familiar with BASIC's byte vectors you can think of 'addr' as the base of a vector, and of X as containing the subscript, e.g.

```
addr?7 = 12
```

is equivalent to

```
LDA #12
LDX #7
STA addr, X
```

5.2 String types

Two examples of the use of indexed addressing are given below, both involving strings. There are two string types available for use in BASIC and assembler; ATOM strings and Microsoft strings. An ATOM string is a string of characters terminated by a RETURN character. The name which identifies the string is preceded by a dollar ($) sign and the strings can be easily set up in BASIC, e.g.

```
$name = "Fred"
```

ATOM strings must have an area of memory set aside for them. This can be done, as in the examples, by using a DIM statement. The characters making up the string are then stored in the location identified by the name of the string. This is very useful as the address of each character is then also known.

A Microsoft string is a string of characters preceded by a byte which gives the length of the string. In this case, the name of the string has a dollar ($) sign after it. It is more flexible than the ATOM string because it can contain RETURN characters. Its disadvantage is that all the characters making up the string are stored in locations chosen by BASIC, hence the addresses of these are not known.

Example - print inverted-case string

The following program uses indexed addressing to

print out a string of characters terminated by a carriage return (which is represented in the memory by &D), swapping case as it prints out each character.

```
 10 DIM string 256, code 100
 20 oswrch = &FFEE
 30 FOR pass = 0 TO 3 STEP 3
 40 P% = code
 50[OPT pass
 60.enter
 70 LDX #0             Set index to zero
 80.loop
 90   LDA string,X     Get characters from string
100   CMP #&D          Is it end of string ?
110   BEQ return       If so, end
120   EOR #&20         Else invert case bit
130   JSR oswrch       Print it
140   INX              Increment index
150   BNE Loop         If string Longer than 256
160.return
170   RTS              then end anyway
180]
190 NEXT pass
200 END
```

Assemble the program by typing RUN, and then try the program by entering:

```
$string = "Test String"
CALL enter
```

Example-- index subroutine

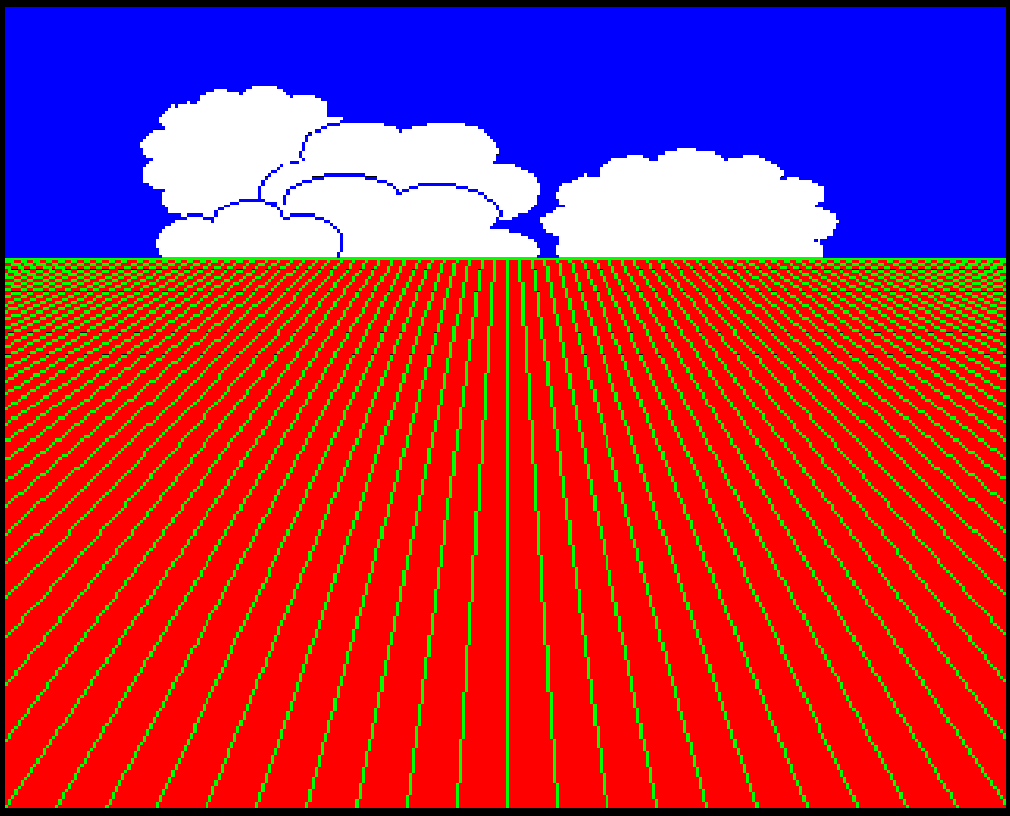
Another useful operation, easily performed in a machine-code routine, is looking up a character in a string and returning its position in that string. The following subroutine reads in a character, using a call to the OSRDCH read-character routine, and saves in '? found' the position of the first occurrence of that character in '$target'. This is exactly the same as the BASIC '?found =INSTR("ABCDEFGH",GET$)'.

```
 0 REM Index Rooutine
10 DIM target 25,P% 100
20 osrdch=&FFE0 : $target="ABCDEFGH" : found = &70
```

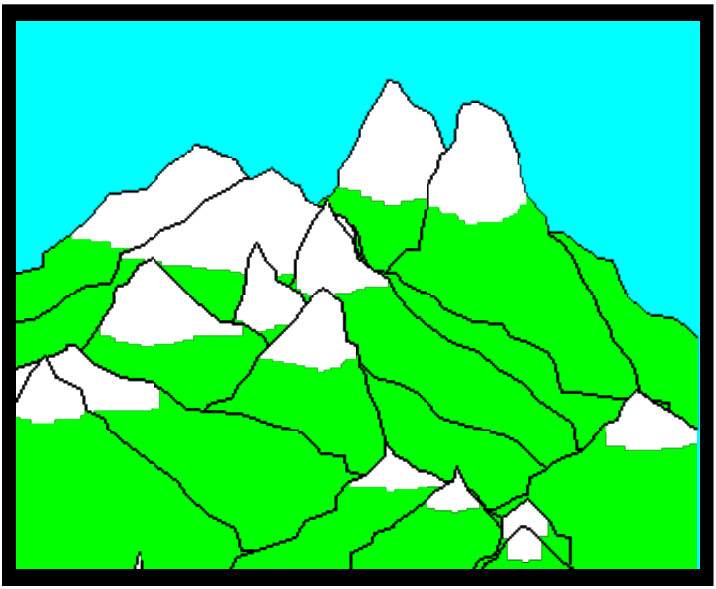
Creative Graphics

on the BBC Microcomputer

JOHN COWNIE



Mountains



This program draws a view of randomly-generated snow-capped mountains as shown in the photograph above. The colours are chosen at random, and although this can produce some ridiculous effects, it can also produce colour schemes that are reminiscent of the subtle hues of an alpine landscape.

The largest most distant mountains are drawn first, and then 'closer' mountains are put on top of these obscuring them where they overlap. This technique for eliminating hidden lines, by drawing from the back and then over-plotting, is useful in many applications. Although at first sight it may seem to be rather slow and extravagant, the alternatives are considerably more complicated and probably not much faster. The method to use will obviously be governed by the nature of the object being drawn, but the 'pasting on top' approach, as used here, gives an easy solution for irregular objects.

The technique relies on the ability to fill in areas of colour rapidly. For line drawings a mask around the foreground object is filled in black (or whatever the background colour is) and then the object is drawn on top of the mask.

When the entire picture is complete the program will wait for any key to be pressed before starting again.

Each mountain is drawn as follows:

1 Choose a random point (X_PEAK%,Y_PEAK%) to be the summit.

2 Choose two values X_SLOPE% and Y_SLOPE% to determine the slope of the right-hand side of the mountain.

3 Draw in steps down the mountain-side by adding random x and y values, determined by the slope, to the current position. The space below the line drawn is filled in with the logical colour 2. If we are still near the top of the mountain the area below the edge down to the snow-line is filled with logical colour 3.

4 The side is followed until it goes off the screen.

5 The method above is repeated for the left-hand side.

6 The mountain is now complete.

MOUNTS

```
    0 REM Mountains
   20 MODE1
   30 VDU19,0,6;0;
   40 VDU5
   50 FOR MOUNTAIN%=900 TO 0 STEP -60
   60 X_PEAK%=RND(1200)
   70 Y_PEAK%=MOUNTAIN%+RND(50)
   80 FOR SIDE%=0TO1
   90 X_SLOPE%=RND(40)+20
  100 Y_SLOPE%=RND(20)+30
  110 MOVE X_PEAK%,Y_PEAK%
  120 X%=X_PEAK%:Y%=Y_PEAK%
  130 REPEAT
  140 IF SIDE%=0 THEN X1%=X%+RND(X_SLOPE%) ELSE
X1%=X%-RND(X_SLOPE%)
  150 Y1%=Y%-RND(Y_SLOPE%)
  160 SNOW_LINE%=Y_PEAK%-Y1%/5-50:GCOL0,2
  170 MOVEX1%,Y1%: PLOT85,X1%,0: MOVEX%,
0:PLOT85,X%,Y%
  180 IF SNOW_LINE%<Y1% THEN GCOL0,3:MOVEX1%,Y1%:
PLOT85,X1%,SNOW_LINE%:MOVEX%,SNOW_LINE%:PLOT85,X%,Y%
  190 X%=X1%:Y%=Y1%
  200 GCOL0,1:DRAWX%,Y%
  210 UNTIL POINT(X%,Y%)=-1
  220 NEXT SIDE%
  230 VDU19,RND(3),RND(8)-1;0;
  240 NEXT MOUNTAIN%
  250 A=GET:GOTO50
```

CHAPTER 2
Trigonometry

Scale drawings

Seldom can we directly measure the heights of tall buildings, hills, trees, etc. One way to find the height of a building or tree is to stand away from the object. Now measure the angle between the horizontal and the highest point of the object (using a clinometer, which is just a glorified protractor), then measure the distance between you can the object. By making a scale drawing the height of the object can be readily estimated. See **Figure 1**.

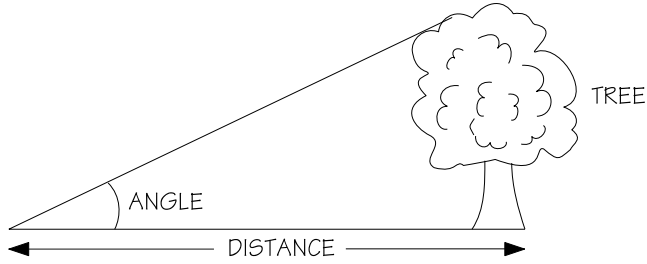


Figure 1.

You couldn't use the same technique to measure the height of a mountain peak which is miles away and covered in clouds. The clouds would get in your way, and you couldn't measure the horizontal distance. An instrument such as a tellinometer would help. This uses radar to locate the top of the mountain. It also measures the angle and distance between you and the top. A scale drawing would provide a way of calculating the height of the mountain. See **Figure 2**.

As a further example suppose we wanted to find the width of a large pond or lake; see **Figure 3**.

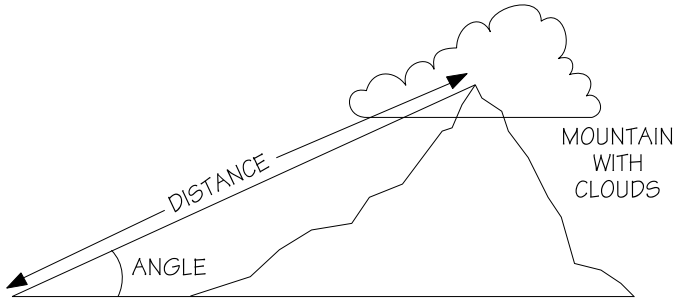


Figure 2.

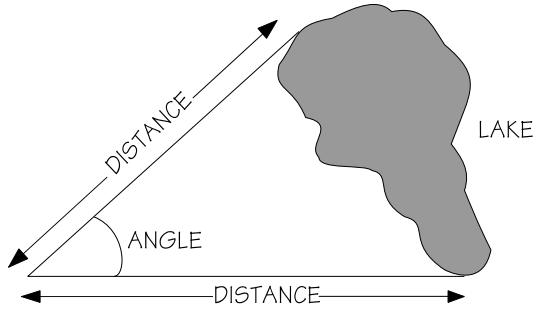


Figure 3.

A scale drawing could be produced from the measurements made, and the required distance estimated.

Here is a related example. A navigator is at a certain position A. He is 150km due west of city B and 188 km from city C. The angle between the two cities is 23 degrees measured from his position. How far apart are the two cities? Again, a scale drawing could provide the answer.

Although scale drawings will provide answers to the problems mentioned above, they are rough and ready. And they cannot always be practicably or accurately reproduced. An alternative approach is to do the necessary calculations by trigonometry using your computer.

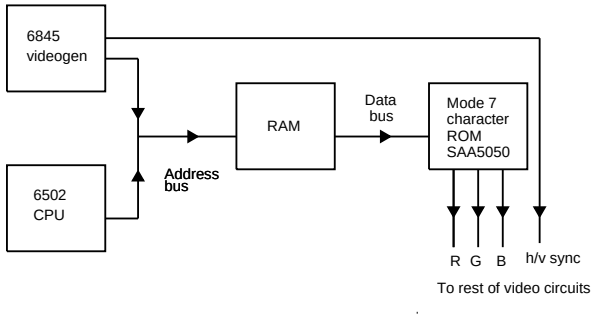


Fig. 1.3. Simplified block diagram of video sections in mode 7.

The interfaces

This is in fact a heading under which to gather together a wide range of different circuits! Some of these, such as the sound generator, the user port and the A to D convertor, for example, are dealt with at length in other chapters but it is worth producing a summary of all the interfaces circuits inside a standard BBC Micro. One thing that all the interface circuits have in common is that they use the 1K of address space not used by the MOS ROM.

The interfacing circuits within a standard Model A BBC Micro are:

1. Cassette and RS423 serial.
2. VIA(Versatile Interface Adaptor) - A. This is a parallel interface looking after internal devices such as the keyboard and the sound generator.
3. The 1 MHz extension bus.
4. The tube.

In the standard Model B machine we have to add:

5. VIA-B - a parallel interface that looks after two external ports, the centronics printer and the user port.
6. An A to D convertor (a mPD7002) that can be used as a general purpose measuring device or with joysticks.

There are other interface circuits that can be added to the BBC Micro beyond even these six, such as the disc interface, but these are of less

general interest and will be discussed in Chapter Nine. We will now take a look at each of the above interfaces, apart from the A to D convertor which is dealt with in detail in Chapter Six and is so separate that it adds little to our understanding of the overall machine.

The cassette and RS423 interface

Every BBC Micro comes equipped with a cassette interface. The interface also doubles as a general purpose serial interface. It is true that owners of the Model A cannot use this serial interface but this is only because the two buffer chips that provide the power to drive the serial output to the RS423 standard are missing. (The RS423 standard is simply an improved version of the older and better known RS232 or V24 standards. For our purpose it may be considered entirely compatible with both.)

The cassette interface on the BBC Micro relies on two major components. The first is a 6850 ACIA (AsynChronous Interface Adaptor) which is responsible for changing data from a parallel to a serial format and vice versa. This is all that is necessary for the RS423 interface (apart from the aforementioned buffering). However, the cassette interface works by recording two audio tones corresponding to the binary zeros and ones in the serial bit stream

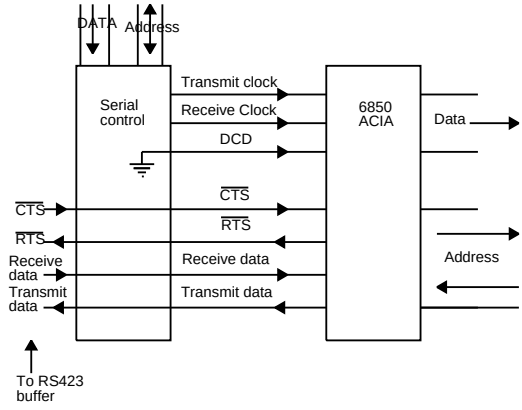


Fig. 1.4. Serial Interface set-up for RS423

THOUGHT GAMES

HIGHER/LOWER
by R.Bailey

This program is based on the popular TV game, 'Play your cards right', where you have to predict whether the next card in a sequence will be higher or lower than the previous one.

Instructions
You begin with a credit of 250 points, and a target of 2500. You can bet between 50 and your current credit balance at each turn.

At the first, fourth and seventh card, you are offered a choice of changing the card. If you reach your target at the ninth card, you will begin a new round with your credit added to your total score. If your credit runs out or you don't reach the target, you are offered a new game.

Programming techniques
The design on the reverse of the cards is generated by PLOT commands in the procedure PROCdraw. Only one user defined character (line 80) is used in this game, and its purpose is to create a figure '10' as a single character. This means that the number fits neatly on the cards in the same way as the lower single-digit numbers.

```
   10 REM Higher / Lower
   20 REM by R.Bailey
   30 REM BEEBUG
   40 REM VERSION P 1.0
   50 :
   60 ON ERROR GOTO 1230
   70 X=RND(-TIME)
   80 VDU23,224,206,219,219,219,219,219,206,0
   90 MODE5:DIMA(12),B$(9),S(12):@%=&00000908
  100 BS=0
  110 VDU28,4,16,19,0:VDU19,128,4,0,0,0:VDU19,2,0,0
,0,0
  120 RT=0:CLS
  130 C=250:MB=50:AIM=2500
  140 F=0:PROCtitles:RESTORE:FORJ=1TO9:READX,Y:PROC
draw:NEXT
  150 PROCset:PROCpicture:RESTORE:F=1
  160 FORJ=1TO9:READX,Y:PROCdraw:PROCshow:IFJ=9THEN
200
  170 IFC<MB J=9:GOTO210
  180 IFJ=1ORJ=4ORJ=7THENPROCchange
  190 PROCbet:PROCupdate:PROChigherlower
  200 NEXT
  210 CLS:RT=RT+C:IFC>=AIM THEN330
  220 IFRT>BS THENBS=RT
  230 IFC<MB THEN310
  240 PRINTTAB(0,2)"You have failed"
  250 PRINTTAB(1,4)"to reach the"
  260 PRINTTAB(4,6)"TARGET"
  270 PROCbestscore
  280 PRINTTAB(0,8)"ANOTHER GAME?"
  290 PRINTTAB(4,10)"Y/N":A$=GET$
  300 IFA$="Y"THEN120ELSE CLG:PRINTTAB(0,10)"SEE YO
U SOON":TIME=0:REPEAT:UNTILTIME=200:MODE6:END
  310 PRINTTAB(0,2)"You have failed"
  320 PRINTTAB(2,4)"miserably":GOTO270
  330 PRINTTAB(0,9)"CONGRATULATIONS"
  340 FORS2%=1TO3
  350 FORS%=1TO12
  360 SOUND1,-15,(S%*4)+200,2
```

3-D MAZE

In this program by Dave Kelsall, you are in a maze (whose dimensions you determine) trying to get through it to the exit. At every turn you see the scene in front of you, and you'll find it quite uncanny how 'solid' and 'real' the maze can seem. If you're in trouble, the 'help option' will provide a bird's-eye view of the maze.

When you first run the program, you'll be asked to enter the width of the maze first, then the height. You can enter any integer between two and 12 for the width, and between two and 10 for the height.

The program length is awesome, but we assure you the effort of entering all of it will be amply repaid.

Here are a couple of sample mazes as generated by the program:

```
              3-D MAZE
Enter size of maze - Width 8
                     Height 8
Thinking-Do you want to see a plan?Y/N
+__+__+__+__+__+__+__+  +
!  !  !  !  !  !  !  !  !
+__+__+__+__+__+__+__+__+
!  !  !  !  !  !  !  !  !
+__+__+__+__+__+__+__+__+
!  !  !  !  !  !  !  !  !
+__+__+__+__+__+__+__+__+
!  !  !  !  !     !  !  !
+__+__+__+__+  +  +__+__+
!  !  !  !  !  !     !  !
+__+__+__+__+  +__+  +__+
!  !  !     !  !     !  !
+  +__+  +  +  +  +__+__+
!  !  !  !  !     !  !  !
+  +  +  +  +__+  +  +  +
!  !        !           !
+__+  +__+__+__+__+__+__+
Type L=Left R=Right F=Forward H=HELP!!
```

```
             3-D MAZE
Enter size of maze - Width 5
                     Height 5
Thinking-Do you want to see a plan?Y/N
+  +__+__+__+__+
!  !  !  !  !  !
+  +__+__+__+__+
!     !  !  !  !
+__+  +__+__+__+
!  !     !  !  !
+  +__+  +__+__+
!  !  !        !
+  +  +  +__+  +
!  !  !  !     !
+__+__+  +__+__+
Type L=Left R=Right F=Forward H=HELP!!
Press Space Bar to proceed


   10 REM     3-D MAZE
   20 REM By Dave Kelsall, St Albans
   30
   40 CLEAR:CLS
   50 ON ERROR RUN
   60 MODE 7
   70 M1=5
   80 A1=1:A2=2
   90 PRINT''''TAB(13);"3-D MAZE"'''
  100 REM**GET MAZE SIZE**
  110 INPUT "Enter size of maze - Width
"H
  120 INPUT TAB(21)"Height "V
  130 PRINT;"Thinking-";
  140 IF H>12 OR H<2 OR V>10 OR V<2 THEN
 PRINT"These dimensions are excessive!"'
:CLEAR:GOTO 110
  150 REM**DIM MAZE ARRAY**
  160 DIM W(H,V+2),V(H,V+2)
  170 Q=0:Z=0:X=RND(H)
  180 REM**SAVE MAZE ENTRY POINT**
  190 C=1:W(X,1)=C:V(0,0)=X:C=C+1
  200 R=X:S=1:GOTO 270
  210 REM**START TO BUILD MAZE**
  220 IF R<>H THEN 260
```

Chapter Two
Ant Hill

Ant Hill is a simple but effective program that involves the animation of a number of objects. The basic idea of the game is to guide a man through tunnels that belong to an ant colony with the aim of reaching and destroying the nest of eggs located at the deepest point. The difficulty of this task is increased by having to work within a time limit and by having to avoid soldier ants positioned at each level of the tunnels. Thee major problem in implementing a game of this sort lies in animating a number of objects, the man and all the soldier ants, at the same time. As well as dealing with this particular problem this chapter also develops some of the standard methods that will be used without further comment in subsequent chapters.

The game design

Before starting to write any program it is a good idea to try to specify, in as much detail as possible, what it should do. Games programs are slightly different from other applications in that it is usually not possible to give an accurate outline of the final game before at least part of it is implemented. The reason for this is that it is very difficult to predict how elements of a game will work without trying them out. Even after writing a large number of games programs it is still difficult to predict the overall effect of combining different elements from existing games to produce a new one. However, it is still worth working out what you expect a game to do before you start writing any of it, for the simple reason that it gives you time to get any major changes out of your system!

The design of Ant Hill (and many other dynamic games) falls nuturally into three parts:

1. the background graphics

2. the moving characters and the rules of movement
3. the consequences of winning and losing

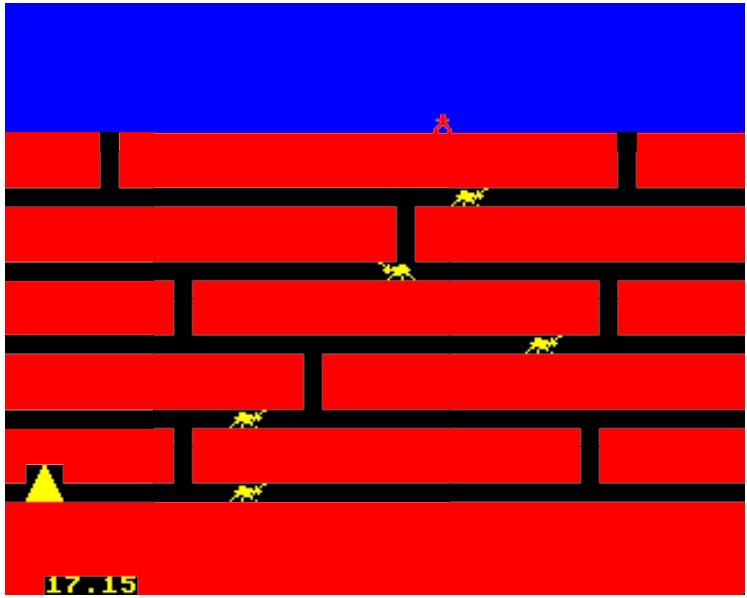


Fig. 2.1.

The background of Ant Hill consists of a number of horizontal tunnels connected by vertical shafts (see Fig. 2.1). The reason for using a mixture of one and two shafts to connect the tunnels is that it promises to give the game more variety of strategy. When only one shaft connects two levels then there is only one route available and playing the game becomes a matter of timing, but when there are two shafts the player has the opportunity of choosing which one to go down. The general layout of the background suggests that at least three colours are going to be needed one for the tunnels, probably black, one for the earth, probably red and one for the sky above, almost certainly blue. This suggests that either a four- or a sixteen-colour mode needs to be used. A choice of the sixteen-colour mode would, however, restrict us to a horizontal resolution of only 20 printing columns and, as Ant Hill is a game that depends on a great deal of horizontal movement, the four-colour mode seems a better choice.

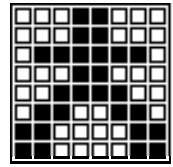
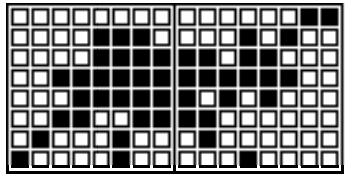


Fig. 2.2. Graphics character for man

The man shape is easily impelemented as a single user-defined graphics character (see Fig. 2.2). Using a single character means that the tunnels and the shafts that connect them only now have to be one character wide. However, the ant shapes are much more difficult to implement in a single eight by eight dot character because they are fairly long and thin. The solution is to use two user-defined graphics characters (see Fig. 2.3). The fact that the ants only move horizontally along the tunnels, and never up or down the shafts, means that even though they are composed of two characters the tunnels and shafts still only need to be one character wide.



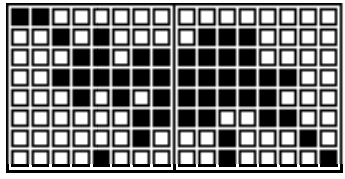


Fig. 2.3. Graphics characters for ants

The rules of movement for the ants are easy to define. Each level will have one ant confined to that level able to :move along the tunnel. If an ant happens to arrive the current of the man character then the

Reading the Listings

Note that all programs in this book have been printed out using the Program Lister program (see pages 18-25). The program formats the listings so that they are easier to read than would otherwise be the case.

One of the most useful features of the BBC Micro's red function keys is to allow entry of teletext control codes and user-defined graphic symbols. These are used by pressing one of the keys at the same time as pressing either SHIFT or CONTROL. The codes produced by these keys produce some weird effects when printed out, so to avoid confusion, it is generally not advisable to print these. The Program Lister program gets round this by converting the codes into unambiguous forms which can be easily printed and understood. The actual output is a direct representation of the keys which need to be pressed to produce the effect desired.

There is another common problem when entering program listings from a book or magazine. It is often hard to tell how many spaces are to be included within a string. The Program Lister counts the spaces in the program for you, and if there are more than two in a row, it outputs the number of spaces required, rather than just printing a vast empty string. One or two spaces are printed directly. The output from the program we've mentioned is enclosed within square brackets, so you'll know that anything within these brackets refers to information produced by the Program Lister and must be entered with care. Here's how to work out what they mean.

[fsn]	means	shift + function key 'n'
[fcn]	means	control + function key 'n'
[chn]	means	character with ASCII code n (see below)
[spcn]	means	n spaces

[chn]

Normally, these codes are not available from the keyboard, but you can program any of them directly into one of the function keys. For example, if the character with code 255 is required (that is, the program refers to ch255) simply type:

> VDU 255

Then after the character appears enter:

> *KEY 0 'character here'

The 0 can be, of course, the key of your choice. (Once the character has appeared, use the edit keys to 'bring it down' into the quotes.)

OS 0.10

All the programs in this book were written on OS 1.20. Most programs will work without modification, but programs with user-defined graphics will need their character codes changed. On OS 1.20, user-defined characters (before explosion) are defined in the ASCII code range 128 - 159 inclusive, but in OS 0.10 they are in the range 224 - 255. Therefore, the codes will need to have 96 added to them.

Also, you'll discover that using the function key in conjunction with SHIFT or CONTROL will not work, so the codes will have to be copied into the keys using the method described above, and the keys are unshifted. Some operating system calls are not supported in OS 0.10, so we've avoided using these as much as possible.

Three of the programs in this book (Mandala, Pontoon and Play Your Cards Right) will not fit on the computer if you have discs. To get them to fit, you need to disable the DFS. You do

this by entering:

```
*BUILD DWNLOAD
   1 *KEY 0 *TAPE|MFORA%=0TO(TOP-PAGE)ST
EP4:A%!&E00=A%!PAGE:NEXT:PAGE=&E00|MOLD|
MRUN|M
   2
Escape
```

The routine is now on your disc for you to use later on. Before you try to load the long program, enter *EXEC DOWNLOAD and then load your program. Press f0 and your program will run as normal.

produces (a tilted square, bottom leftish). Finally, try

```
PROC_RESTART : PROC_MOVETO(200,-300,0)
PROC_SQUARE(200)
```

to see how it is possible to use non-basic turtle commands. Try to let the squares dance by using PROC_SQTURN.

In this routine I and A$ are local to the procedure: I is used as a loop counter, and A$ is used as a means to produce early termination (the F key is pressed). For up to 600 times the cursor (or turtle) moves forward a distance I (without drawing), draws a square of side I, and then the turtle turns through 30 degrees. The keyboard is checked by INKEY$(0) to see if a key is pressed (saved in A$); if the key was an F, then ENDPROC else the loop counter is incremented.

The routine is activated by

```
PROC_RESTART : PROC_SQTURN
```

and it runs remarkably quickly. If you want to slow it down, put a pause in the INKEY$, eg INKEY$(20), but this does not slow down the drawing of the squares — it just increases the time between squares (see UG page 276, for INKEY$). This is a tediously predictable routine — it is the same every time. The predictability is shown in **Icon 1.1**.

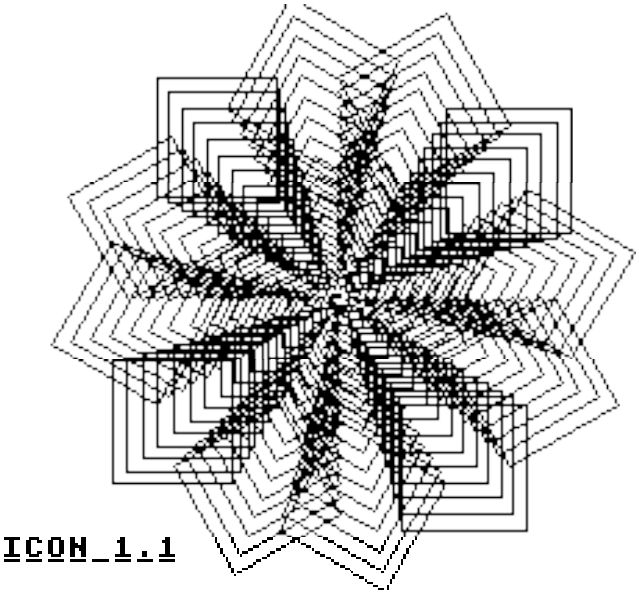
Note that Figures are not computer output, they may be diagrams or tables which are there to assist in the understanding of the text. An Icon — which is ‘an image, picture, or representation’ according to the dictionary — is an exact copy of a display on the screen, it is a screen dump onto a printer.

Need a rest? Take Five.

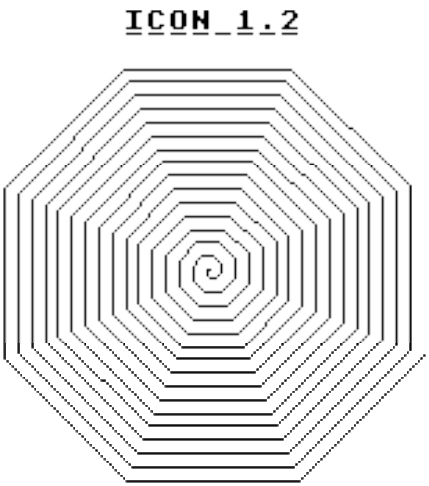
An unsquare dance

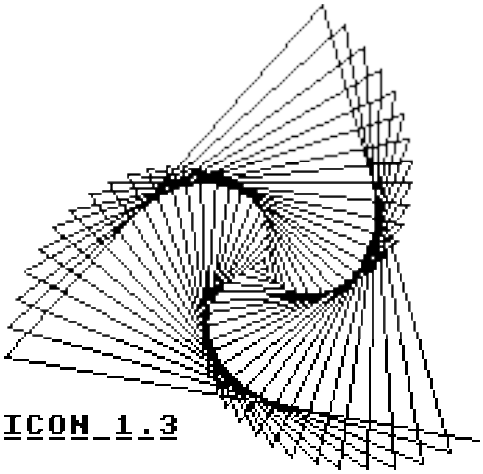
I walk forward a certain, fixed, distance and turn through a certain, changing, angle: what happens?
Solution: see PROC_INSPIRALI.

A$ is local again, tedious but safe, and the routine repeats until the F key is pressed. The distance is kept constant (ie SIDE), but the angle (ANG) is incremented (by INC) at each pass through the indefinite loop (ie REPEAT. . .UNTIL. . .). This routine produces a vast array of unpredictable results, which, once known, are completely predictable. It is named PROC_ANSPIRALI because it is an INward SPIRAL, coded Iteratively. Iteration means, as explained in the previous chapter, that the control of the procedure is by a loop mechanism — in this case REPEAT. . .UNTIL.

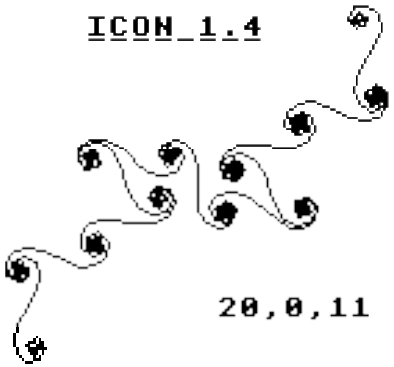


An outward SPIRAL is shown by PROC_OUTSPIRAL, in which the angle remains constant and the distance increases, and is what we normally mean by a spiral. **Icons 1.2** and **1.3** show two examples of outward spirals for varying values of the fixed angle.





An outward spiral — as the name spiral suggests — keeps on spiralling outwards, but an inward spiral does nothing as common as that. Icons 1.4 to 1.7, are examples of four highly different shapes. Try to watch what happens as the plotting unfolds: if it helps to slow down the process, change the value of the parameter in the INKEY$.



```
580 DY=SQR(ABS(DX*DX-R*R))
590 DRAW X+DX,Y-DY
600 NEXT DX
610 ENDPROC
```

Fig. 2.14. Program to draw random size circles

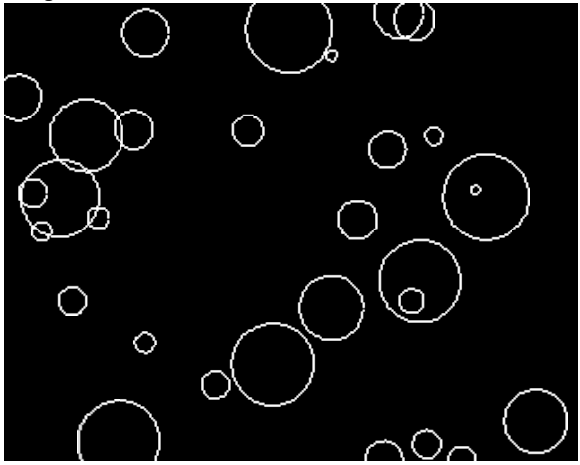


Fig. 2.15. Display of random size circles produced by program of Fig. 2.14.

With this routine the number of calculations depends upon the size of the circle, and it will be seen that the larger circles take a noticeable time to draw. This is because the computer has quite a lot of calculations to carry out. The square root function itself is rather slow in BASIC. Things could be speeded up slightly by calculating R*R outside the drawing loop and using the result in the calculation for DY. This saves some multiplication operations but the overall calculation is still quite slow. If we want to draw circles faster we will need to look at other ways of calculating the points around the circle.

The rotation method

A different approach to the calculation of the X,Y values for a circle is to base them upon the angle through which the radial line is rotated at each step. In this case the new values for X and Y are calculated from the previous values rahter than from the radius and the total angle.

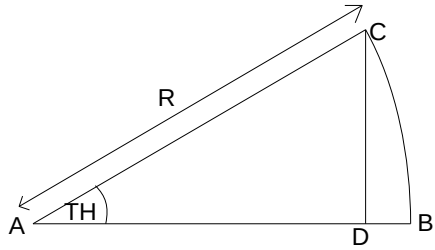


Fig. 2.16. Rotation of a point from X axis.

If we look at Fig. 2.16 the initial value of Y is zero, so that only the original X value, which also happens to be equal to R, affects the results. Here we get

```
X1 = X * COS(TH)
Y1 = X * SIN(TH)
```

Now consider the situation where the radial line is vertical and is moved through angle theta. This is shown in Fig. 2.17. Here the initial value of X is 0 and only the Y value affects the results. In this case the value of Xl is negative since the X point has been shifted to the left of the line where X=0. Here we get the results

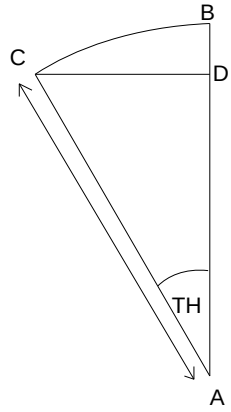


Fig. 2.17. Rotation of a point from the Y axis.

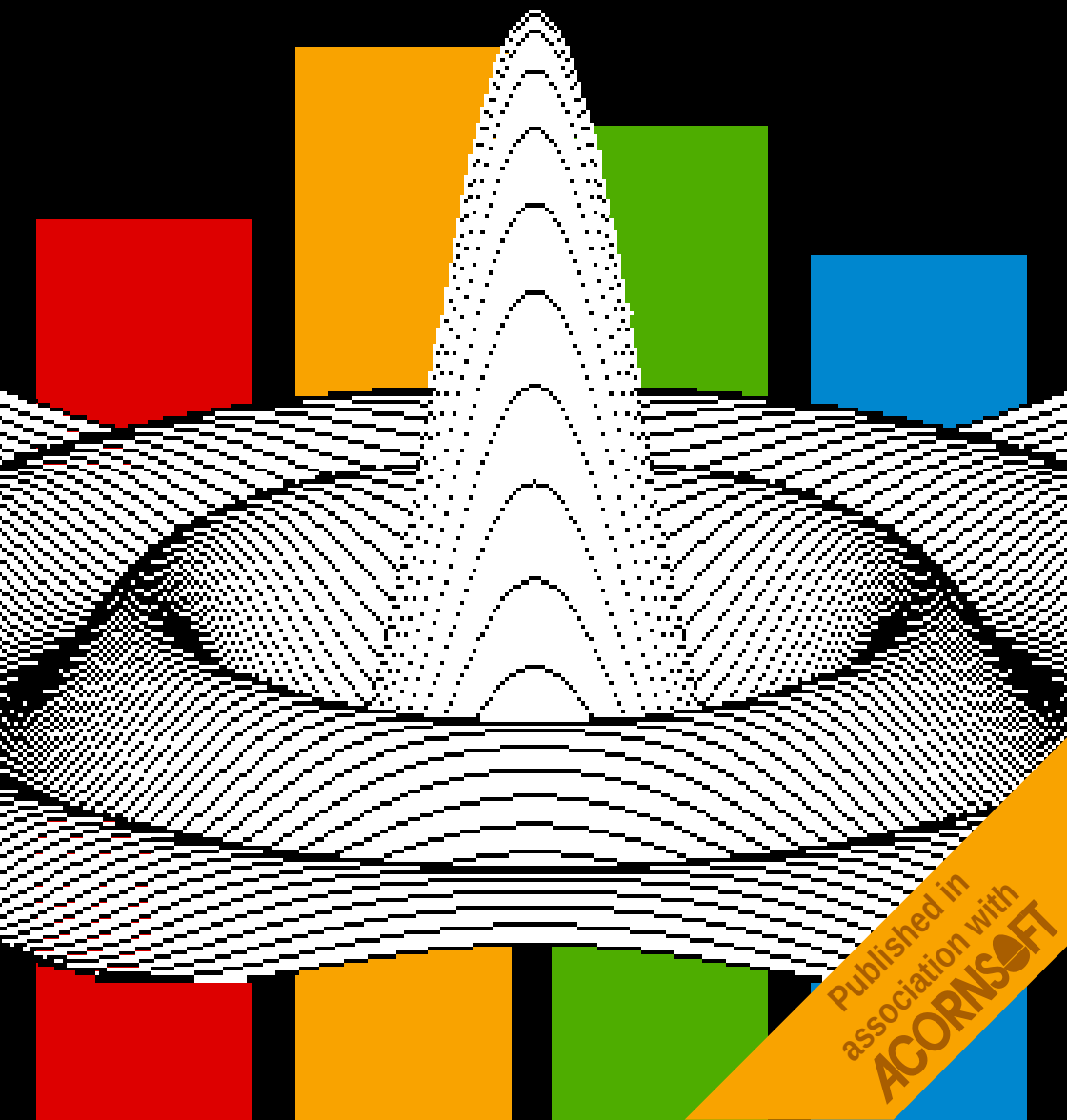
```
X1 = -Y * SIN(TH)
Y1 =  Y * COS(TH)
```

If we combine these two results we can produce a general expression for calculating Xl and Yl for any initial values of X and Y. The two new equations are

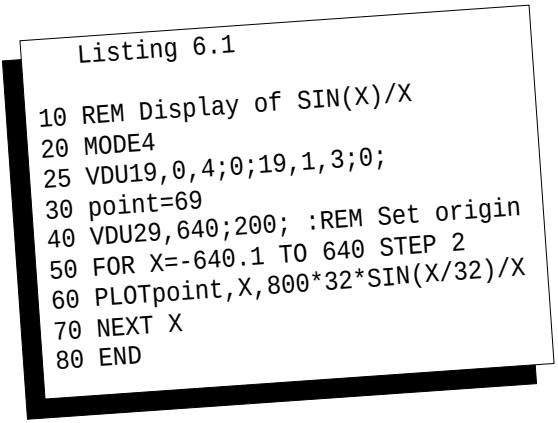
```
X1 = X * COS(TH) - Y * SIN(TH)
Y1 = X * SIN(TH) + Y * COS(TH)
```

Neil Cryer, Pat Cryer and Andrew Cryer

Graphics on the BBC Microcomputer



Published in association with ACORNSOFT



```
Listing 6.1

10 REM Display of SIN(X)/X
20 MODE4
25 VDU19,0,4;0;19,1,3;0;
30 point=69
40 VDU29,640;200; :REM Set origin
50 FOR X=-640.1 TO 640 STEP 2
60 PLOTpoint,X,800*32*SIN(X/32)/X
70 NEXT X
80 END
```

As you see, the program uses the point-plotting version of the PLOT statement. Line 40 uses the following special version of the VDU statement, which allows the origin for any future graphics to be altered to X,Y:

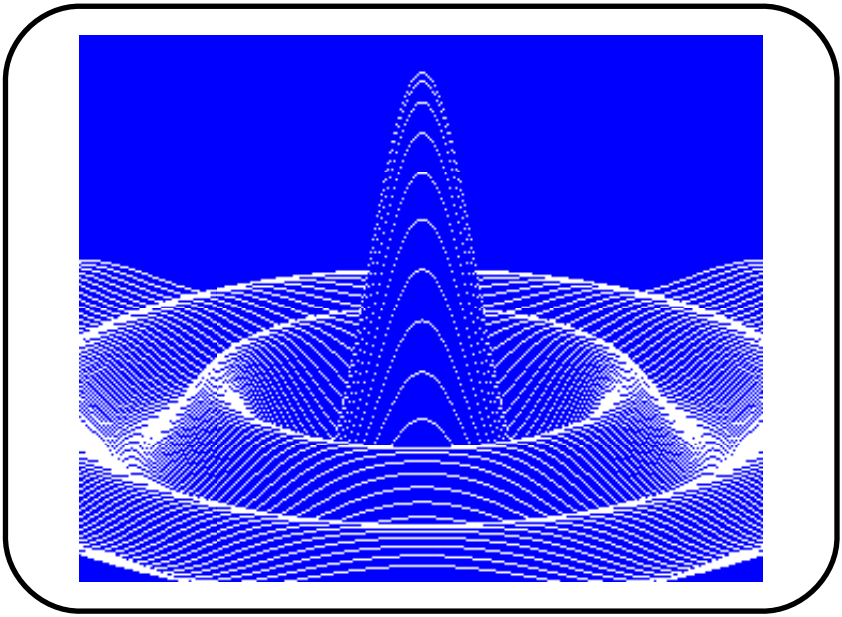
VDU29,X;Y;

In the program this sets the origin for graphics to the point 640,200. This is because the function is symmetrical about X=0; so we felt the program would be clearer if the values of X ran from -640 to +640. The scaling takes place in line 60. The 800 enlarges the plot to fill most of the screen and the X/32 controls the number of bumps on the curve. Such scaling is usually best done by starting with an intelligent guess, displaying the resulting plot and then adjusting the scaling.

6.2 Activities

This activity helps you to appreciate the importance of scaling on the appearance of a display.

i. Enter the program of Listing 6.1 and run it.



Screen Display 6.2

ii. Try altering the overall size of the display scaling factor 800 in line 60.

iii. Try altering the number of 'bumps' in the display by varying the scaling factor 32 in line 60.

iv. In line 50 the value of X is purposely set to start at -640.1 rather than at -640 exactly. Investigate why, by altering to -640.

v. Try adding STEP 4 to line 50 in order to speed things up.

6.3 Drawing the surface

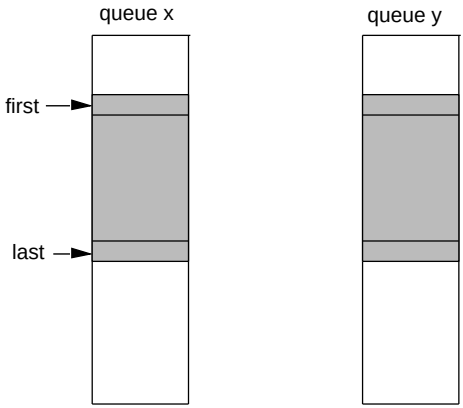
We now show how to use SIN(R)/R to produce the symmetrical three dimensional ripple surface of Screen Dispiay 6.2. The height of any point on the surface is dictated by the value of the function at that point. There is a central, main 'bump' just as there is for the two dimensional view of Screen Display 6.1.

```
480   first=(first+1)MOD500
490 ENDPROC
```

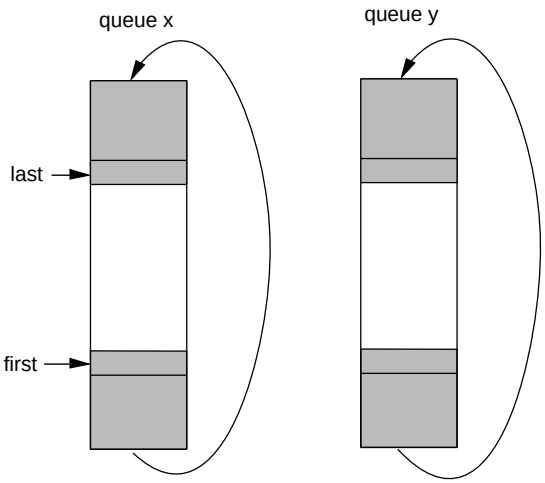
PROCfillfrom is initiated from a start point and that start point is coloured and added to a queue (by calling PROCfilll. PROCfillfrom then repeatedly takes the first point from the queue and examines each of the neighbouring N, S, E and W points (by calling PROCfill for each of these points in turn). Each time PROCfill is called, it colours the point it is given (if it is not already coloured) and adds that point to the end of the queue. Adding a point to the queue in this way ensures that it will subsequently be removed from the queue and its neighbours examined.

The reason the queue is made a FIFO is to prevent it becoming too large. If for example we made the queue an ordinary stack (LIFO or last in first out), as you may see suggested in computer graphics textbooks, it would gradually fill up and would run out of memory.

For the queue, we use two arrays, one for x-coordinates and one for y-coordinates. Two variables indicate the positions of the 'first' and 'last' items in the queue.



The arrays are treated as circular so that when the end of the queue reaches reaches the end of the arrays, the queue is 'wrapped around' and continues into the space that is now free at the start of the arrays.

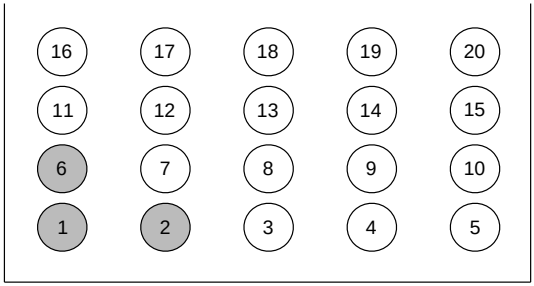


PROCfillfrom repeatedly takes the next point from the queue until the queue is empty. The photograph shows the algorithm in the course of filling. Note that the 'wavefronts' are diagonal. This is a consequence of using a FIFO queue in this particular context.

An illustrative sequence of how the algorithm works in detail is now given for a simple rectangular region. The start point is the bottom left hand corner.

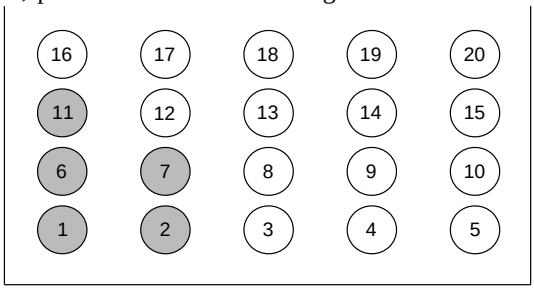
pixel 1 is filled and added to the queue

1st cycle of REPEAT loop in PROCfillfrom
pixel 1 is removed from queue and neighbouring points examined



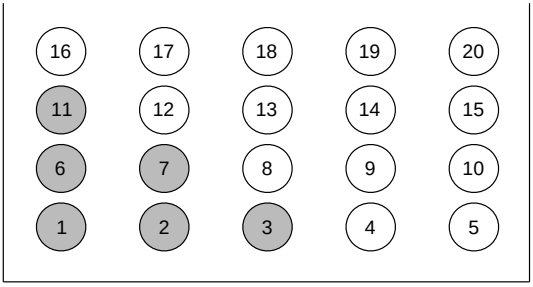
queue is now 6, 2
pixels 6 and 2 are filled

2nd cycle, pixel 6 removed and neighbours examined.



queue is now 2, 11, 7
pixels 11 and 7 are also filled.

3rd cycle, pixel 2 removed and neighbours examined.



queue is now 11, 7, 3
pixel 3 is filled

Addison-Wesley Software

BBC MICRO & ELECTRON

MANUAL

HALLAM

GRAPHITO

DIGITALLY REMASTERED EDITION

draws it on a net.

3) Type **RUN** and press **RETURN**

Program prelude

The prelude that is loaded with the module is:

```
1 MODE4 : HIMEM=HIMEM-1260-2340
2 PROCinitialisememory(4,TRUE,TRUE)
```

This can be altered to make best use of the computer's memory in the following circumstances:

```
1 MODE4 : HIMEM=HIMEM-2340
2 PROCinitialisememory(4,FALSE,TRUE)
```

if you are not using motifs

```
1 MODE4 : HIMEM=HIMEM-1260
2 PROCinitialisememory(4,TRUE,FALSE)
```

if you are not using alphabets

Procedures

The following procedures are available

PROCalphaslice
PROCcharnet
PROCchardesign
PROCdrawandscale
PROCfore_to_back_col
PROChshear
PROChtext
PROCinitialise
PROCload
PROCloadalpha
PROCloadscreen
PROCnet
PROCreflectx
PROCreflecty
PROCrestore_fore_col
PROCrotate
PROCsavescreen
PROCscale
PROCstretch
PROCvshear
PROCvtext

Details of how to use each of these procedures are given later in the manual

Using 2DMOD3

There are exactly 17 ways in which an asymmetric motif can be arranged to form a two dimensional network pattern. These are known as wallpaper groups.

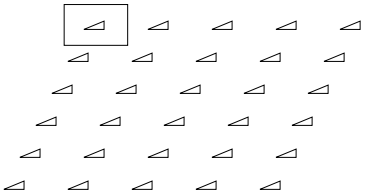
Each can be generated in this system by generating a motif cluster using DEFPROCnetmotif together with an appropriate call to PROCnet. The groups are summarized below using an asymmetric triangle as a motif. Each group is given together with a suggested recipe for PROCnetmotif. In each case adjustments in the x and y parameters of PROCdrawandscale may be necessary to gain the correct symmetry. Note that this method of using elementary motifs to make a motif cluster is not the only to way to proceed. With group 1 7, for example, you could use the generation scheme for group 1 provided you started with a motif that possessed the appropriate rotational symmetry.

The motif 'cluster' generated by PROCnetmotif is boxed in each illustration.

Group 1

This is the basic network group and simply requires a motif to be placed at each point on the net. The motif definition should be;

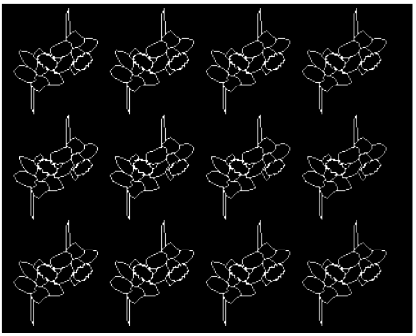
```
1000 DEFPROCnetmotif(x,y,scale)
1010 PROCload("MOTIF")
1020 PROCinitialise
1030 PROCdrawandscale(......)
1040 ENDPROC
```

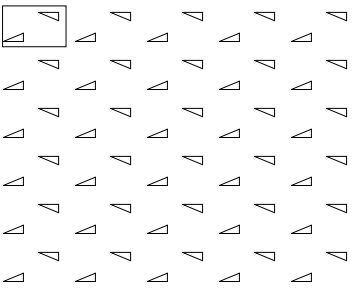


Group 2

This group involves a reflection about the x axis together with an appropriate displacement.

```
1000 DEFPROCnetmotif(x,y,scale)
1010 PROCload("MOTIF")
1020 PROCinitialise
1030 PROCdrawandscale(......)
1040 PROCreflectx
1050 PROCdrawandscale(......)
1060 ENDPROC
```

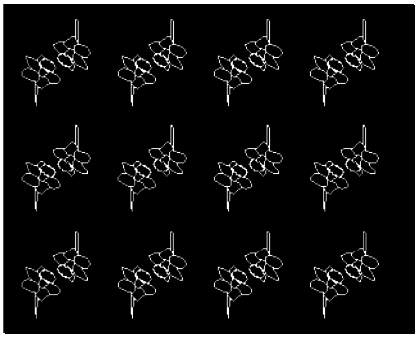


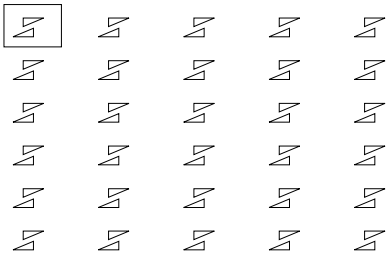


Group 3

The motif cluster in the group is formed by a 180° rotation.

```
1000 DEFPROCnetmotif(x,y,scale)
1010 PROCload("MOTIF")
1020 PROCinitialise
1030 PROCdrawandscale(......)
1040 PROCrotate(180)
1050 PROCdrawandscale(......)
1060 ENDPROC
```





Group 4

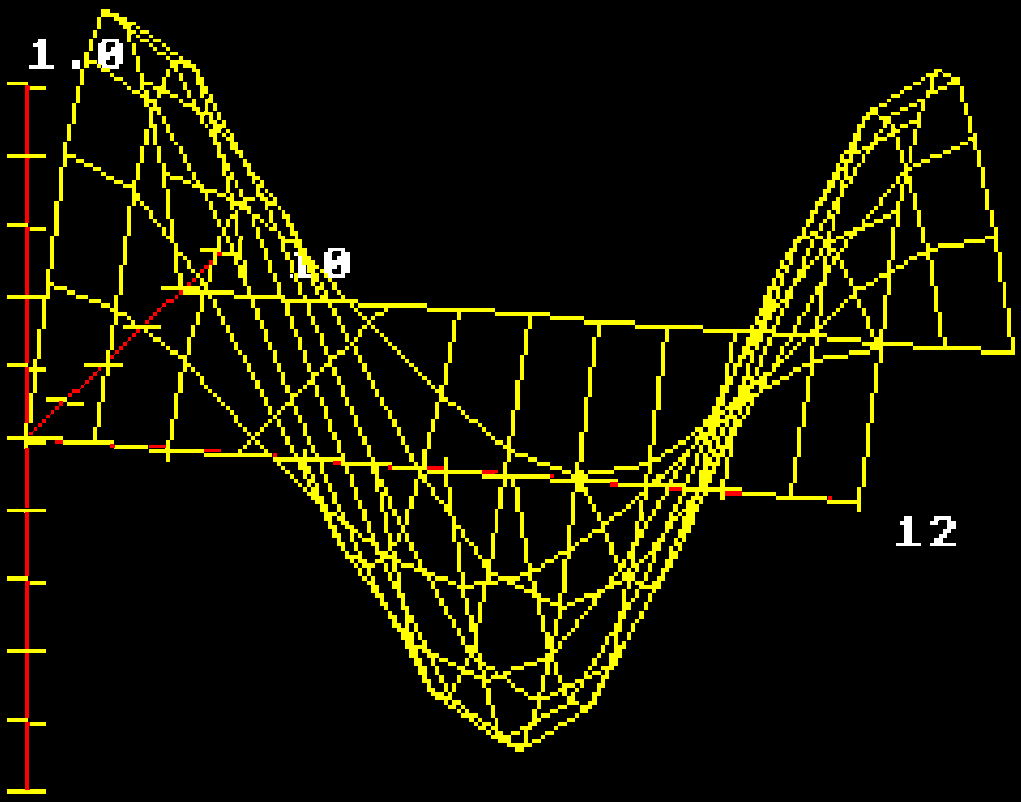
This motif cluster is formed from four motifs

```
1000 DEFPROCnetmotif(x,y,scale)
1010 PROCload("MOTIF")
1020 PROCinitialise
1030 PROCdrawandscale(......)
1040 PROCrotate(180)
1050 PROCdrawandscale(......)
1060 PROCinitialise
1070 PROCreflecty
1080 PROCdrawandscale(......)
1090 PROCrotate(-180)
1100 PROCdrawandscale(......)
1110 ENDPROC
```

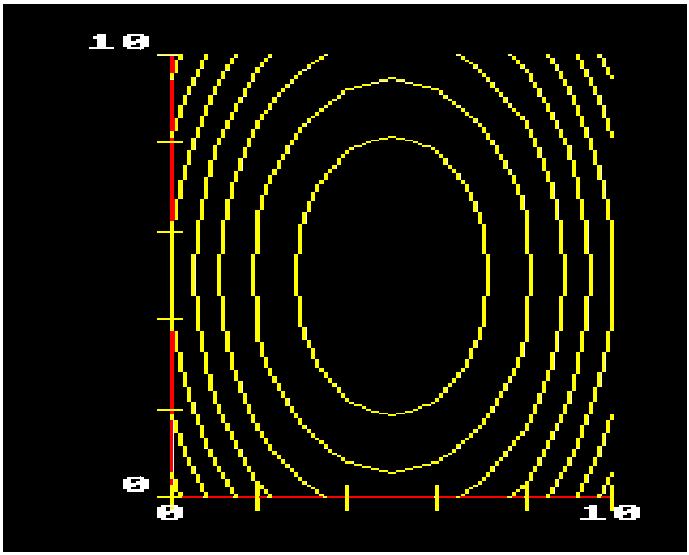
Graphs and Charts

on the BBC Microcomputer

ROBERT D. HARDING



Example Program




```
 10 REM L2-C2X1
 30 DIM £W(10,10)
 40 FOR I%=0 TO 10:FOR J%=0 TO 10
 50 X=(I%-5):Y=(J%-5)
 60 £W(I%,J%)=X^2+Y^2/2
 70 NEXT:NEXT
 80 REM now call PLOT£CN2D
 90 MODE5:PROC£CN2D(5,10,10,10)
100 END
```

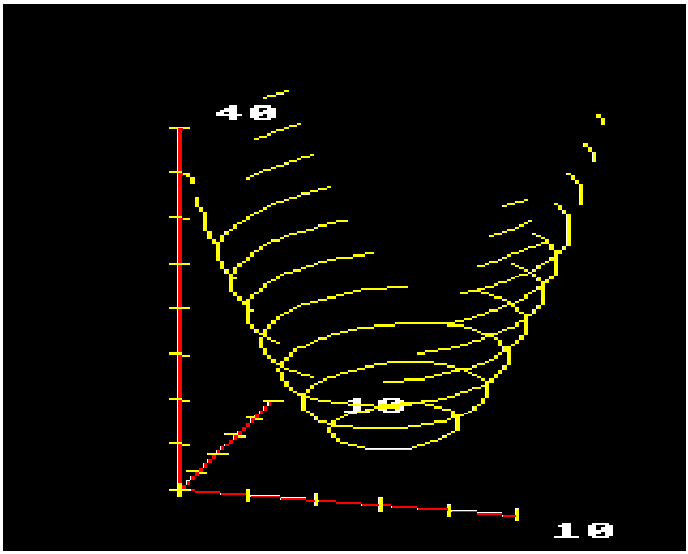
L2-C3 Complete Perspective Contour Map Plotter

PROC£CN3D(M,IM,JM,N) - Requires Ll-3D

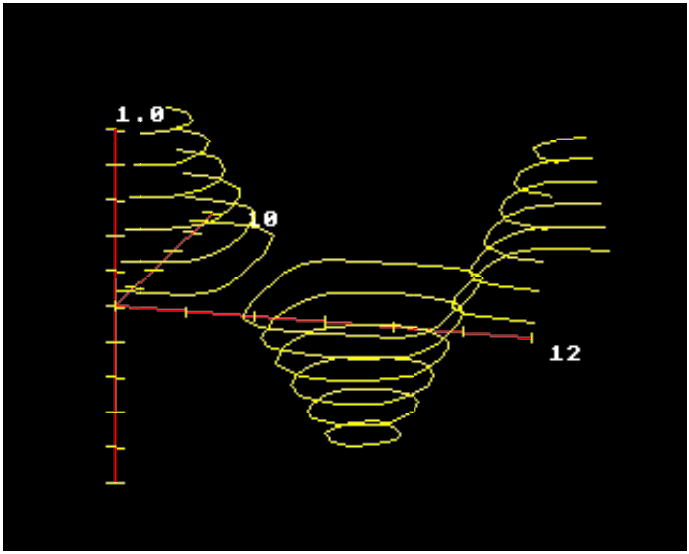
This procedure works in the same way as L2-CN2D except that the contours are drawn in perspective view according to height.

The array £W(I,J) must be dimensioned and set exactly as for L2-CN2D, then the call PR0C£CN3D(M,IM,JM,N) will draw the contours.

Example Programs




```
 10 REM L2-C3X1
 30 DIM £W(10,10)
 40 FOR I%=0 TO 10:FOR J%=0 TO 10
 50 X=(I%-5):Y=(J%-5)
 60 £W(I%,J%)=X^2+Y^2/2
 70 NEXT:NEXT
 80 REM now call PROC£CN3D
 90 MODE5:PROC£CN3D(5,10,10,10)
100 END
```




```
200 REM L2-C3X2
230 DIM £W(12,8)
240 REM
250 REM set w = cos(x).sin(y)
260 FOR J=0 TO 8: B=SIN(J*PI/8)
270 FOR I=0 TO 12
280 £W(I,J)=B*COS(I*PI/6)
290 NEXT
300 NEXT
310 REM
320 REM now call PLOT£CN3D (12 contrs)
330 MODE1:PROC£CN3D(1,12,8,12)
340 END
```

The Procedures

PROCanimate

What it does: Displays an animated graphics character at any required position on the screen.

Formal parameters: *R*, the number of times the animation sequence is to be repeated.

X,*Y*, the TAB coordinates of the position on which the character is to be displayed.

C, the ASCII code of the first of the pair of user-defined characters.

Local variables: *J* and *T*, the loop indices.

Actual parameters: *repeats*, *tabx*, *taby*, *code*.

Listing:

```
   10 MODE 4
   20 INPUT "Enter the code, 224 to 254 
"code
   30 VDU 23,code,196,71,69,127,124,124,
72,108
   40 VDU 23,(code+1),101,71,66,127,124,
124,72,108
   50 INPUT "Enter the X coordinate "tab
x
   60 INPUT "Enter the Y coordinate "tab
y
   70 INPUT "Enter the number of repeats
```

```
 "repeats
   80 PROCanimate(repeats,tabx,taby,code
)
   90 END
  100 REM ************************
  110 DEF PROCanimate(R,X,Y,C)
  120 CLS
  130 LOCAL J,T
  140 FOR J=1 TO R
  150 PRINT TAB(X,Y)CHR$(C);
  160 FOR T=1 TO 1000:NEXT
  170 PRINT TAB(X,Y)CHR$(C+1);
  180 FOR T=1 TO 1000:NEXT
  190 NEXT
  200 ENDPROC
```

How it works: The procedure displays a pair of user-defined characters alternately at the same location on the screen. This produces the illusion of movement. The characters will have been defined previously (in the calling program). The action of the procedure takes place in a loop (lines 140 to 190) which is repeated R times. At each repetition, the first of the two characters is displayed, followed by a pause (line 160). Then this character is replaced in the same position by the second character. A second pause follows, and the loop is then repeated.

When the procedure ends, the second character is left on display.

Calling program: This procedure works in any graphics mode. Mode 4 is used for this demonstration. You are first asked to enter the ASCII code for which you required the first of the two characters to be defined. The second character will be defined for the code number following this. Lines 30 and 40 then define two characters using VDU 23 statements. The characters given in this example program show a dog turning its head and wagging its tail.

Next you are asked to enter the two TAB coordinates at which the dog is to be displayed. X should be between 0 and 39, and Y shoujd be between 0 and 3 I. Suitable values are 20 and IS which place the dog at the centre of the screen. Line 80 calls the procedure, which clears the screen and displays the dog in action.

At the end of the procedure, the second character is left on the screen. In your own program you may want to remove it, either by printing a space at the same TAB position, or by clearing the whole screen. Or perhaps you may prefer to leave it there, motionless, ready

to be animated again at some later stage in the program.

Variations: There is no end to the variety of graphics designs that can be used with this procedure. In any single program you could use to 15 different pairs of characters, displayed at different parts of the screen.

If you wish to make the character move faster or slower, alter the 1000s in lines 160 and 180 accordingly

Associated routines: PROCmoveacross, PROCmovedown.

PROCblankline

What is does: Clears the whole of the screen line that the cursor is on, returning the cursor to the beginning of that line.

Formal parameters: None.

Local variables: X, the number of character per screen line in the current mode.

Actual parameters: None.

Listing:

```
 10 CLS
 20 INPUT "TEXT "text$
 30 PROCblankline
 40 END
 50 REM ************************
 60 DEF PROCblankline
 70 LOCAL X
 80 X=(?&352+256*?&353)/?&34F
 90 VDU 13,11
100 PRINT STRING$(X," ")
110 VDU 11,11
120 ENDPROC
```

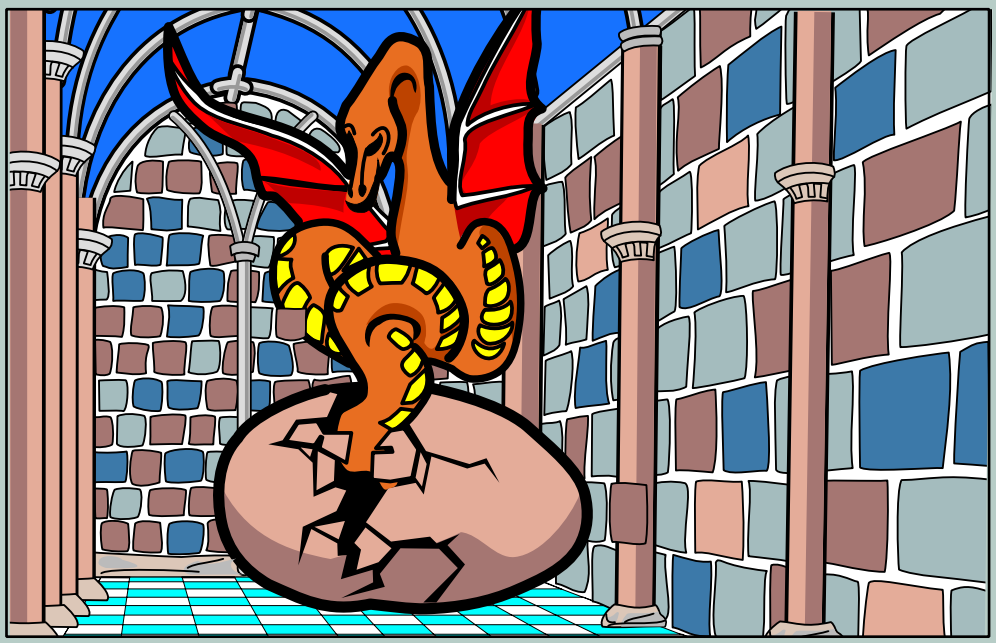
How it works: Line 80 finds *X* by reading two values from memory. The first of these is the number of bytes needed for storing one screen line. The second is the number of bytes needed for storing one character. Dividing one value by the other gives *X*, the number of characters in a screen line. Line 90 is a VDU statement which sends the

DIGITALLY REMASTERED EDITION

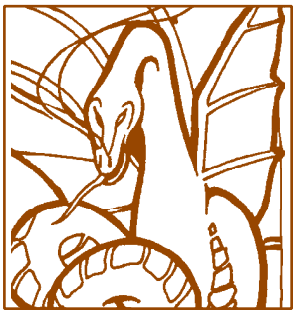
How To Write ADVENTURE GAMES

for the BBC Microcomputer Model B and Acorn Electron

Peter Killworth



PENGUIN ACORN COMPUTER LIBRARY



2

CREATING A ‘HACK-AND-SLASH’ GAME: ‘CAVES´

2.1 The plot

In this part of the book we will investigate a simple adventuring game in order to demonstrate how to develop a database structure, handle commands, and so on.

As a model we will look at the bones of a more complex but truly excellent Adventure called Sorceror’s Cave (the publishers are now Gibsons, and its writer is Terence Donnelly).

The game presents a team of people (initially just one man) involved in an exploration of a vast cave system, which will change from game to game. You begin in an entrance cave just underground, and may explore in any one of six directions: north, east, south, west, up and down (though not all areas have exits in each direction). Exploration reveals a three-dimensional grid of caves and passages, which should be mapped in order to avoid getting lost.

In the caves lie treasures of various values, and denizens. The latter are both ordinary people, like yourself, and mythical creations such as giants, dragons, and so on. Often the denizens are guarding treasure or blocking a route you wish to take. You have the choice, as you would in real life, of deciding to leave well alone (a

good choice where dragons are concerned!), fighting them (and gaming the advantage of surprise) or approaching them to sec if they're prepared to explore with you (which will strengthen your party for later encounters). In the latter case, the denizens' leader will determine whether he likes the look of you, doesn't care one way or the other, or wants to fight you (whereupon the denizens get the advantage of surprise). Some denizens are more friendly than others . . .

Fighting is carried out by the program, not by the player, and involves a comparison (or weighing up) of fighting strengths. Each character in the game has a fighting strength; weak characters like hobbits have little strength, whereas fearsomely strong characters like dragons resemble a mobile army. The fighting strengths of all denizens present are added together and pitted against the fighting strengths of up to three of your party (since caves are awkward places, not too many people can muscle in on the fight!). There is a bonus of one for whichever side has the surprise advantage. To each of these numbers is added a random dice throw (i.e. a number between one and six). The team with the highest number kills one of the other side. In the event of a draw, you are deemed to be still fighting. This gives you the chance of running for an exit, or continuing to fight.

Only if you kill all the denizens, or if they join your party, can you pick up the treasure in then area. There are more treasures (and more denizens!) in the lower levels of the caves; thus the surface levels are safer if less rewarding.

Each area is either a cave with something inside it, or an empty passage. The type of area, its exits, and contents if any, are determined randomly the first time the player attempts to enter. Thus the player may be in a passage with a north exit but be unable to use it because the area to the north doesn't have a southern entrance.

Finding and using an 'up' staircase on a level of the caves just underground will take you out of the caves, and finish the game. Things are seldom that sample because caves and passages get blocked easily, and there are two further random events which can ruin your plans. One is an earthquake, which will destroy the area you were just in, and render it impassable. The other, a

trap, is a precipitous drop one level deeper into the cave system. In either of these cases, there is no possibility of retreat should you encounter any denizens.

That's roughly the plot. In the original, access to the next cave or passage was determined by revealing a card with a representation of a cave and some exits drawn on it, followed by a number of cards representing Ms contents. In our implementation, this will be replaced by random selection within the computer. However, there will have to be a fair amount of book keeping so that the player may backtrack and rediscover caves (and where appropriate, contents) that have already been mapped.

2.2 Planning the game – the game logic

Having decided on the plot of the game, the next thing is to organise its logical flow, turn by turn. I strongly recommend doing this in English, or a quasi-English most programmers know as 'pseudo-code'. What we we do us write the program in readable English, but in terms that are converted, with relative ease, to BBC BASIC.

Since BBC BASIC is highly structured, and we aren't going to be short of room for this program, we can set up a very structured program; please bear this in mind as we proceed.

First of all, when writing pseudo-code,

GET THE LOGICAL STRUCTURE FIRST

then figure out later how to program it. Only if the programming is clearly beyond your abilities should you redesign the structure!

We'll assume that outside the main program loop there will be some initialisation, dimensioning, screen mode choosing, etc., and concentrate now on the recurring logic. As he takes a turn, the player may be in a normal, 'what shall I do now?' situation, or he may be 'still fighting' from a previous turn. Obviously the latter will take precedence over the former. So we begin our pseudo-code with:

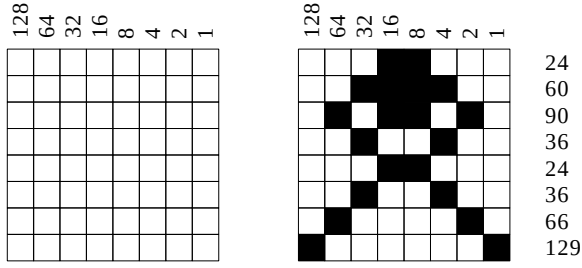
IF PLAYER IS FIGHTING, MAKE THE AREA UNSAFE, SET FIGHT BONUS TO ZERO, AND ASK PLAYER IF HE WISHES TO CONTINUE TO FIGHT OR TO LEAVE. IF CONTINUING, FIGHT AND GO TO END

CHAPTER 6

Character Graphics

In this chapter we will be taking a look at one of the most important parts of programming games: Graphics characters. We have already used many of these without bothering to worry about how they are made up or how they work. Let's take a look at one of the routines that sets up a graphics character. For example, look at the routines that define aliens. As you can see, they are all very similar, in fact the only differences are in the lines starting with REM and the line starting with VDU. The line starting with REM, if you remember, just means REMark or REMinder, so after REM you can write anything you like and the program will ignore this statement - it's just there to help us humans understand the program.

Next we'll deal with the VDU lines. You may have noticed that there are always eight numbers in each VDU statement which are separated by a space from the first two. This has just been done so that we can see where the last eight numbers start because it is these numbers which define our character. The first number in the statement is 23, and this lets the computer know we want to redefine a character. The second number is the code of the character which we wish to redefine, and then we come to those eight pieces of data. Why eight? Well, it's all to do with the way in which our little characters are made up. They are drawn on an eight by eight grid of squares. We have to draw our character on a grid like this, and remember, you can only use whole squares, not parts - it's all or nothing.



Above is a blank grid and next to it we've drawn a new alien so that you can see how to turn him into a set of numbers. If you look again, you can see the row of numbers across the top of the grid. Each square in the grid has a value, and that is given by the number above it, so to get the number for each line, you start at one end, and if the square is blank you move to the next, and if the square is blacked in you add it to your total. The first line is hence:

16 + 8 = 24

and the rest of the lines are as follows:

32 + 16 + 8 + 4 = 60
64 + 16 + 8 + 2 = 90
32 + 4 = 36
16 + 8 = 24
32 + 4 = 36
64 + 2 = 66
128 + 1 = 129

This is just like binary arithmetic, which isn't too surprising. Binary is a number system based on zeroes and ones and it is the only number system that your micro-computer understands directly. It appears to understand decimal numbers but that is only because there is a program inside your micro-computer that converts everything - even the words - into binary numbers.

Now you know enough to be able to design your own graphics characters, so you can have different aliens or players. You can even design new shapes for use in the background. What we really need to know, though, is how the routine actually works. So let's go back over the program in greater detail.

The REM statement we have already dealt with, and we know that REM is short for REMinder or REMark.

The VDU statement contains a list of numbers which are the decimal equivalent of the pattern of binary numbers. Binary arithmetic is used beacuse it is easy for a computer processor to recognise one of two states, on or off, making it possible to represent only the two numbers, 1 and 0. Memory is divided into bytes - there are a maximum of 65536 of these in your computer

and each byte is further subdivided into 8 bits. Each bit (short for binary digit) can have the values 0 or 1 and as you move along the byte twoards the left, each bit is worth double that of the one before it - hence the sequence of numbers at the top of our graphics grid:

128,64,32,16,8,4,2,1

If each bit was set to 1, the number held in that byte would be 255 decimal, which is represented by 11111111 in binary. These bits are copied from memory in a special area (at address &C00) and stored there for future reference. When you PRINT a character the data is copoied to the appropriate screen position and a point of light appears where each 1 bit is and a dark point where each 0 bit is. This means that when you PRINT out your character now you will get a little monster or whatever shape you designed.

Following this text is a utility program (a utility program is one that helps you to design and ceate other programs) to help you make up your own characters and change them around without using yards and yards of paper and wearing out lots of pencils. (After all, what's a computer for if not to make life easier?) The instructions for using it are as follows:

Your position in the grid is shown by an asterisk. To move it around, use the cursor keys (the arrow keys).

When you reach a square you want to change - either from black to white or white to black - press C and it will change.

After you are satisfied with the design, press S and the program will ask you in which character you wish to save your design. If you wished your design to replace character 128 then you would simply enter 128 followed by RETURN, and the character at the top of the screen would be replaced by your new character and you would be able to see what it looked like at proper size.

This is a longish program so be careful when you key it in.

Pan

PERSONAL

COMPUTER

COMPUTER NEWS LIBRARY

JEFF AUGHTON

INVALUABLE UTILITIES for the ELECTRON

'The complete programmer's toolkit – essential programming aids for your micro'

DIGITALLY REMASTERED EDITION

Utility 4:
Disassembler

Description

One of the best features of the Electron is the built-in Assembler. This makes it possible to write (source) code using 6502 mnemonics and symbolic labels which are then translated into machine (object) code by the Assembler. A disassembler performs the reverse process and is absolutely essential for anyone writing – or even merely interested in – machine code.

The output from this disassembler is similar to that of the Assembler except that:

i) labels are not given

ii) ASCII equivalents are supplied

iii) branch instructions are provided with a direction and the absolute destination address

During disassembly, any invalid op-codes are assumed to be single byte instructions and are replaced by '???'

As it can be difficult to follow disassembled code on the screen, you are given the option of sending output to the printer when you first run the utility.

At this point you may feel inclined to turn to another section on the grounds that you are not yet *au fait* with machine code. If not, why not?? You have the idea! machine on which to learn and there are now numerous books and articles on the subject. The Electron has been well designed to make machine code programming as painless as possible and it is well worth making an effort to learn. Throughout the book I assume that you are prepared to have a go at understanding the assembler routines even though you may prefer to be reading BASIC.

If you are still somewhat apprehensive about machine code a good place to start is to type in this disassembler and use it to look at how other people write programs. There are many machine code programs available for the Electron and you can learn a great deal just by studying them with the disassembler. In addition, several of our utilities are written in machine code, although you do not need to be familiar with machine code to get them to work or to understand the principles underlying their operation.

Use

Normally it is best to LOAD the disassembler at the usual value of PAGE and then set PAGE=PAGE+&C00 so that programs may be LOADed and RUN without affecting it. To use the utility, reset PAGE to the correct value and then RUN. Select 'N' in response to the question 'Disassemble to printer?', and enter the address (in hex) at which disassembly is to commence. For starters, try an address somewhere in BASIC, i.e. between &8000 and &BFFF – you will then see how the experts write machine code. If the result of this is a Whole mass of ???'s (invalid op-codes) it is because you have landed in the middle of a data table rather than executable code. BASIC contains several such tables, so try a different address if you find one.

Pressing key 'A' will generate one line of output and if you hold it down lines are produced at the rate of about three per second. If you press ESCAPE, you are returned to the question 'Start address?' so that you may continue disassembly from a different point. To exit the program, you should press ESCAPE in response to this question.

This approach will not be so successful if the program you want to disassemble has to be LOADed in at PAGE for correct operation (as might be the case for, say, a video game). Any absolute addresses within the program would be displaced by an amount: (actual LOAD address – true LOAD address), making the code very difficult to follow. The simple solution to this is to Low the disassembler in a different place – for example, near the top of memory to make room for the intended disassemblee (if I may coin a word). Before doing so you should switch to MODE 6 and reserve at least 12 pages (&C00 bytes) for the routine.

Alternatively, the disassembler can be modified to include an offset facility so that it can stay where it is and pretend that the code it is disassembling is actually located somewhere else. We will look at the offset facility in the Extensions section below.

```
   10 REM DISASSEMBLER
   20 MODE 6:@%=1
   30 VDU 19,1,3,0,0,0
   40 INPUT "Disassemble to printer (Y/N
)",A$
   50 IF A$="Y" vdu%=2 ELSE vdu%=15
   60 ON ERROR GOTO 890
   70 VDU 3,28,0,1,39,0,12
   80 INPUT "Start address: &"A$
   90 pc%=EVAL("&"+A$):start%=pc%
  100 ON ERROR GOTO 920
  110 VDU 28,0,24,39,2,12,vdu%
```

```
  120 REPEAT
  130 IF vdu%=2 OR INKEY(0)=13 PROCline
  140 UNTIL FALSE
  150
  160 REM ONE S/R PER ADDRESSING MODE
  170
  180 RETURN : REM SOME SUBROUTINE!!!
  190 PRINT "A";:RETURN
  200 PRINT "#";pc%?1;:RETURN
  210 GOSUB 230:PRINT ",X";:RETURN
  220 GOSUB 230:PRINT ",Y";:RETURN
  230 PRINT "&";:PROChex(pc%?1):RETURN
  240 d%=pc%?1:to%=pc%+2+d%:X$="+"
  250 IF d%>127 to%=to%-256:X$="-":d%=25
6-d%
  260 PRINT X$;d%;" (";
  270 GOSUB 340:PRINT ")";:RETURN
  280 PRINT "(&";:PROChex(pc%?1):PRINT "
,X)";:RETURN
  290 PRINT "(&";:PROChex(pc%?1):PRINT "
),Y";:RETURN
  300 GOSUB 330:PRINT ",X";:RETURN
  310 GOSUB 330:PRINT ",Y";:RETURN
  320 PRINT "(";:GOSUB 330:PRINT ")";:RE
TURN
  330 to%=(pc%?1)+256*(pc%?2)
  340 PRINT "&";:PROChex(to% DIV 256):PR
OChex(to% MOD 256):RETURN
  350
  360 DEFPROCline
  370 pc%=pc% AND &FFFF
  380 PRINT " ";
  390 PROChex(pc% DIV 256)
  400 PROChex(pc% MOD 256)
  410 PRINT " ";
  420 byte%=?pc%
  430 IF (byte% AND 3)=3 byte%=3
  440 RESTORE
  450 FOR I%=0 TO byte%-(byte% DIV 4):RE
AD code$:NEXT
  460 am%=ASC(code$)-96
  470 mn$=RIGHT$(code$,3)
  480 asc$=""
  490 ex%=-(am%>2)-(am%>9)
  500 FOR I%=0 TO ex%
  510 asc%=pc%?I%
  520 PRINT " ";
  530 PROChex(asc%)
```

Chapter Four
Program Formatters

BASIC's LISTO command allows a limited amount of control in producing formatted listings, inserting spaces to indent loops and structures as required. The two programs presented in this chapter provide an extended formatting option for either BASIC or assembler programs; indeed, the Assembler Formatter was used to produce the clear listing within this book, inserting ten spaces between line number and mnemonic but leaving labels un-indented and clearly separated from the listing.

```
 10 REM * A Basic Formatted Listing *
 20 FOR loop=0 TO 100
 30 PRINT loop : NEXT loop
 40 INPUT "A number" N%
 50 IF N%=10 PRINT"Correct" ELSE PRINT
"wrong"
 60 REPEAT : INPUT "Code" C$
 70 FOR wait=0 TO 1000 : NEXT wait
 80 UNTIL C$="END"

>LIST

 10 REM * A Basic Formatted Listing *
 20 FOR loop=0 TO 100
 30   PRINT loop
     : NEXT loop
 40 INPUT "A number" N%
 50 IF N%=10 PRINT"Correct"
      ELSE PRINT "wrong"
 60 REPEAT
     : INPUT "Code" C$
 70  FOR wait=0 TO 1000
      : NEXT wait
 80  UNTIL C$="END"
```

Fig. 4.1. A BASIC listing with and without the BASIC formatter

The BASIC formatter splits multistatement lines by issuing a carriage return each time it encounters a colon. It also splits IF. .
.THEN. . .ELSE structures in addition to indenting them along with

REPEAT . . . UNTIL and FOR. . .NEXT loops. Figure 4. 1 shows the type of listing the BASIC Formatter is capable of. Now for the programs!

The BASIC Formatter (Program 4.1)

The basic_format procedure assembles its machine code into Page 9 of block zero RAM. This area has a number of uses (in addition to housing our machine code) and is more normally associated with ENVELOPEs 5-16, the speech buffer, cassette and RS 423 buffer. The routine has two entry points - &900 and &928 in this case - and function keys I and 2 have been programmed to call these locations. These two entries simply turn the formatter on and off respectively.

The 'on' entry point (line 1485) first prints the formatter on message before storing the current value of LISTO, found at &1F, in a byte above the program. Its maximum value of 7 is then inserted. The WRCHV vector contents are extracted and saved and the WRCHV pointed to the *format' entry point at line 1521. The 'off' entry, line 1506, simply reverses these procedures. Line 1518 could be changed if required to make the formatter clear the LISTO option each time it is switched off by replacing it with

LDA #0

```
  10 REM *** LISTING FORMATTER ***
  20 PROCbasic_format(&900)
  30 *KEY0 CALL &900|M
  40 *KEY1 CALL &928|M
  50 END
  60 :
1480 DEF PROCbasic_format(addr)
1481 interpreter=&E0A4
1482 FOR pass=0 TO 3 STEP3
1483 P%=addr
1484 [OPT pass
1485 .on
1486           LDX #&00
1487 .next_character
1488           LDA message,X
1489           JSR &FFE3
1490           INX
1491           CMP#13
1492           BNE next_character
1493           LDA &1F
```

```
1494            STA listo
1495            LDX #&07
1496            STX &1F
1497            LDA &20E
1498            STA address
1499            LDA &20F
1500            STA address+1
1501            LDA #format MOD 256
1502            STA &20E
1503            LDA #format DIV 256
1504            STA &20F
1505            RTS
1506 .off
1507            LDX #&00
1508 .next_character
1509            LDA message2,X
1510            JSR &FFE3
1511            INX
1512            CMP #13
1513            BNE next_character
1514            LDA address
1515            STA &20E
1516            LDA address+1
1517            STA &20F
1518            LDA listo
1519            STA &1F
1520            RTS
1521 .format
1522            PHA
1523            CMP #ASC(":")
1524            BNE no_colon
1525            JSR output
1526            LDA #&00
1527            STA byte
1528            STA byte+1
1529            BEQ not_else
1530 .no_colon
1531            LDA #&01
1532            CMP &1E
1533            BNE not_same
1534            LDA #&00
1535            STA byte+2
1536            STA byte+3
1537            STA byte+4
1538 .not_same
1539            CPY #&00
1540            BEQ carry_on
1541 .not_else
1542            PLA
1543            JMP interpreter
1544 .carry_on
1545            LDA &37
1546            CMP #&E7
```

CHAPTER 5
The ENVELOPE Command

The ENVELOPE command is arguably one of the most difficult commands to master in BBC BASIC. Part of the problem lies in the fact that it must be followed by 14 parameters and used in conjunction with the SOUND command. This alone gives us a myriad of possibilities to choose from and the chances of getting things wrong are considerable.

The advantages of knowing what to do when searching for an effect, as opposed to resorting to a trial and effort method, cannot be overemphasised - unless you have a lot of time on your hands; and when does time pass more quickly than when you're programming your computer?

Chapter 7 explores the trial and effort method and how to get the most out of it - with the minimum of effort. This chapter explores the systematic method, one you will find infinitely rewarding once you are able to think of a sound and know immediately how to produce it.

The ENVELOPE command has two separate functions. The first is to control the amplitude of the sound and the second is to modulate the pitch. When a SOUND command is controlled by an envelope, amplitude control is automatically passed to the envelope. In order to produce a sound, the envelope must be configured to do so. Control over pitch is optional and can safely be ignored when experimenting with the amplitude parameters. The Hawaiian Guitar program in Chapter 3 demonstrates pitch control.

The complete ENVELOPE command

Using the notation on page 245 of the User Guide, the ENVELOPE command is followed by 14 parameters and described as follows:

ENVELOPE N,T,PI1,PI2, PI3,PN1,PN2,PN3,AA,AD,AS,
AR,ALA,ALD

The parameter names could have been slightly better chosen but, as these are in common use and many people will be used to thinking in these terms, there is little point in adding further complexities to the situation by introducing new ones. I think of the parameters in these terms:

*N*umber of envelope
*T*ime of each step

*PI*tch 1
*PI*tch 2
*PI*tch 3

*P*itch *N*umber of steps *1*
*P*itch *N*umber of steps *2*
*P*itch Number of steps *3*

*A*mplitude change during *A*ttack
*A*mplitude change during *D*ecay
*A*mplitude change during *S*ustain
*A*mplitude change during *R*elease

*A*mplitude *L*evel for *A*ttack phase
*A*mplitude *L*evel for *D*ecay phase

They may help, or you may have your own mnemonics. The parameters, their ranges and functions are fisted in **Figure 5.1** for easy reference.

Exploration of the two aspects of envelope control will be much easier if they are considered separately and, if the six pitch parameters are set to 0, we can observe the effects of altering the amplitude section.

First, we will see how the loudness contour of a sound can be broken down into sections.

ADSR: the amplitude envelope

ADSR or Attack, Decay, Sustain and Release, has been mentioned in previous chapters in relation to the way in which the volume of a note varies during production. Although the ADSR principle is most commonly used to describe instrument characteristics, the favourite example used to explain it is that of a car approaching us along a straight road. We hear it very quietly at first and it gradually becomes louder until it draws level with us at which point it is as loud as it is going to get. The volume then immediately begins to decrease. If it stops a little further on for the driver to ask directions, the engine volume will remain constant. When it drives off again the volume will gradually fade to nothing. If we plot the volume against time, the resulting graph might well look like **Figure 5.2.**

This example is obviously very coarse and long (in terms of time), but the principle behind the volume variations involved are exactly the same as those which occur when an instrument produces a note. The note envelope, however, will usually be over in one or two seconds, often less.

Figure 5.1

AMPLITUDE ENVELOPE PARAMETERS

PITCH ENVELOPE PARAMETERS

PARA-METER	RANGE	FUNCTION
N	1 to 16	Envelope Number
T		Length of each step in hundredths of a second.
	1 to 127	Pitch envelope auto repeats.
	129 to 255	Pitch envelope does not repeat. T assumes a value mod 128.
PI1	-128 to 127	Change of pitch per step in firs section.
PI2	-128 to 127	Change of pitch per step in second section.
PI3	-128 to 127	Change of pitch per step in third section.
PN1	0 to 255	Number of steps in first sectior
PN2	0 to 255	Number of steps in second section.
PN3	0 to 255	Number of steps in third section.
AA	-127 to 127	Change in amplitude per step during attack phase (heading towards ALA).
AD	-127 to 127	Change in amplitude per step during decay phase (heading towards ALD).
AS	-127 to 0	Change in amplitude per step during sustain phase (heading towards 0).
AR	-127 to 0	Change in amplitude per step during release phase (heading towards 0).
ALA	0 to 126	Target amplitude level AA is aiming for.
ALD	0 to 126	Target amplitude level AD is aiming for.

CHAPTER TEN

SPRITE GRAPHICS

Most arcade games feature a series of animated characters which movearound the screen. One or more of the characters are controlled by the player. On the BBC Micro, the only help the operating system gives to anyone trying to produce such graphics is the provision of theu ser-definable character set. Using VDU23 it is easy to produce eight-by-eight pixel shapes which can be moved around the screen.

However, this system has its limitations. Firstly, the shape producedcan only be in two colours, background and foreground; secondly, eight-by-eight is too small for most purposes; and thirdly, this method is far too slow for a fast action-packed arcade game.

The first and second problems can be solved by combining more than onecharacter to make up an object, but this makes the animation even slower.It is slow because time is taken up by the characters having to be converted from the eight-byte format of the user-defined character to theform in which they are actually stored in the screen memory. Worse still,the way a character is stored varies between the different screen modes.

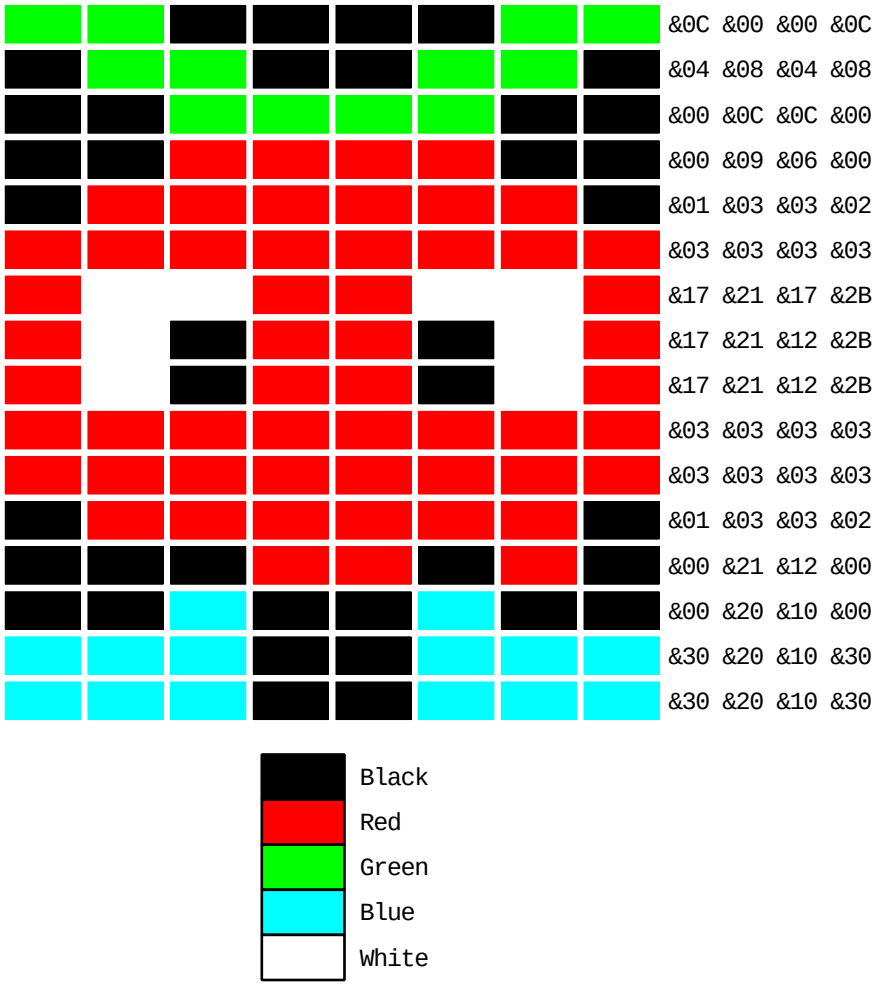
However, as most games will only use one graphics mode, it should bepossible to code the character into the relevant format for thatparticular moe when writing the program. This ready-defined characterwould be able to contain all the colours available in the mode and couldbe any size. This shape could then be stored directly on the screen in afraction of the time taken by the operating system to do the same job. These predefined characters are called SPRITES. As most arcade games workin

Mode 2, I am going to show how to use a complete sprite system from machine code in this mode.

Remember that the methods about to be detailed will not work across the Tube.

Before we embark on a complex machine code routine, we should try an experiment in BASIC − this, as we have seen, is always a good idea when writing machine code routines.

A sample sprite



A BASIC sprite routine We are going to use the sprite shown above as an example. The codng forits storage as a Mode 2 sprite is shown. Because of the way in which the

screen is laid out this coding will only work if the sprite starts on thefirst pixel of a screen memory byte. That is, the furthest left pixel ofthe sprite must be on an even-numbered pixel horizontally. If we wanted toplace it a single pixel to the right or left, we would have to totally re-code it.

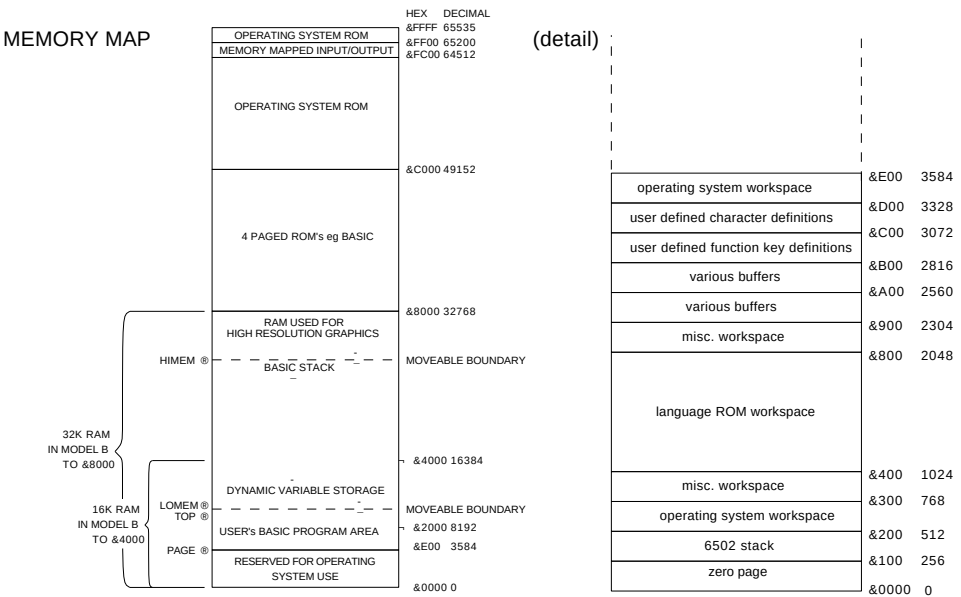
However, we can easily move the sprite left and right two pixels at atime; that way, we are moving it one byte at a time. If we made the spritemove this distance every fiftieth of a second (which is the rate at whichthe image on a TV or monitor is updated), the image would appear to bemoving smoothly.

However, if we wanted the sprite to move slower than that, we wouldeither have to put up with noticing that the sprite jumps two pixels at atime, or we would have to define two sprites, one in each position, andalternate between them. This is common practice in arcade games and veryoften the two sprites are slightly different. For example, it is quiteeffective to use two sprites of a man with his legs in differentpositions. This will make him appear to walk when the sprites are placedalternatey on the screen.

For movement up and down, we need to place the bytes from the shapetable into the screen memory in different positions. As we shall see, thisis not too difficult. We can move the sprite up and down a pixel at a timewithout having to recode the sprite shape table.

The BASIC program below will place our example sprite in the top left-hand corner of the screen. The data statements at the end contain thecoded data for the sprite laid out as above. Remembe, this is an 8-by-16sprite, so it is stored as 4 bytes (= 8 pixels) wide and 16 bytes high.

```
10 MODE2
20 VDU23,1,0;0;0;0;
30 FOR A%=0 TO 1
40 FOR B%=0 TO 7
50 FOR C%=0 TO 3
60 READ D%
70 ?(&3000+A%*640+B%+C%*8)=D%
80 NEXT,,
```


MEMORY MAP
HEX DECIMAL
OPERATING SYSTEM ROM
&FFFF 65535
MEMORY MAPPED INPUT/OUTPUT
&FF00 65200
&FC00 64512
OPERATING SYSTEM ROM
&C000 49152
4 PAGED ROM's eg BASIC
&8000 32768
RAM USED FOR HIGH RESOLUTION GRAPHICS
HIMEM ®
BASIC STACK
MOVEABLE BOUNDARY
32K RAM IN MODEL B TO &8000
&4000 16384
DYNAMIC VARIABLE STORAGE
LOMEM ®
TOP ®
MOVEABLE BOUNDARY
16K RAM IN MODEL B TO &4000
USER's BASIC PROGRAM AREA
&2000 8192
PAGE ®
&E00 3584
RESERVED FOR OPERATING SYSTEM USE
&0000 0
(detail)
operating system workspace
&E00 3584
user defined character definitions
&D00 3328
user defined function key definitions
&C00 3072
various buffers
&B00 2816
various buffers
&A00 2560
misc. workspace
&900 2304
language ROM workspace
&800 2048
misc. workspace
&400 1024
operating system workspace
&300 768
6502 stack
&200 512
zero page
&100 256
&0000 0

4 MEMORY MAP AND MEMORY MAP ASSIGNMENTS

FF00- FFFF
Operating System ROM

FE00- FEFF
Internal memory mapped input/output (SHEILA).

FD00- FDFF
External memory mapped input/output (JIM).

FC00- FCFF
External memory mapped input/output (FRED).

C000- FBFF
Operating system ROM

B000- BFFF
One or more languages ROMs (e.g. BASIC, PASCAL).

4000- 7FFF
Optional RAM on Model B.

0000- 3FFF
Always RAM.

E00
Default setting of PAGE

D80- DFF
Allocated to machine operating system

D00- D7F
Used by NMI routines (e.g. by Disc or Econet filing system)

COO- CFF
User-defined character definitions

B00- BFF
User-defined function key definitions

A00- AFF
RS423 receive, and cassette workspace

900- 9FF
RS423 transmit, cassette, sound and speech workspace

800- 8FF
Miscellaneous workspace

400- 7FF
Language ROM workspace

300- 3FF
Miscellaneous workspace

200- 2FF
Operating system workspace and indirection vectors

100- 1FF
6502 stack

000- 0FF
Zero page

ZERO PAGE

FF
The top bit is set during an ESCAPE condition
F0- FE
Address following detected BRK instruction
FC
User IRQ routine save slot for register A
D0 to FB
Allocated to machine operating system
B0 to CF
Allocated to current filing system
90 to AF
Allocated to machine operating system
70 to 8F
Free for user routines
0 to 6F
BASIC langauge

5 OPERATING SYSTEM COMMANDS

Command	Min abbr.	Meaning
*BASIC	*B.	selects BASIC ROM
*CAT	*.	displays catalogue of files
*CODE	*CO.	allows user to incorporate his own command into operating system command table
*DISC	*D.	selects disc file system
*EXEC	*E.	text files can be used if they were keyboard input
*FX	*F.	OSBYTE calls may be performed directly from keyboard (see *FX calls)
*HELP	*H.	prints version number of operating system
*KEY	*K.	programs user-defined keys (see section on keys and cursors)
*LINE	*LI.	executes machine code at location pointed to by contents of USERV
*LOAD	*L.	loads a section of memory
*MOTOR	*M.	turns cassette motor relay on and off
*NET	*N.	selects network file system
*OPT	*O.	For cassette and ROM filing systems *OPT 0,0 restores default values *OPT 1,0 turn off filing system messages etc.
*ROM	*RO.	selects ROM file system
*RUN	*/	loads and executes a machine code program
*SAVE	*S.	saves section of memory
*SPOOL	*SP.	copies all screen output to named file
*TAPE	*T.	selects cassette filing system (1200 baud)
*TV	*TV	moves picture up and down screen E.g. *TV255 moves display one line down
*|	*|	ignored by computer (used to put remarks in a series of operating system commands)
*/	*/	treated as *RUN

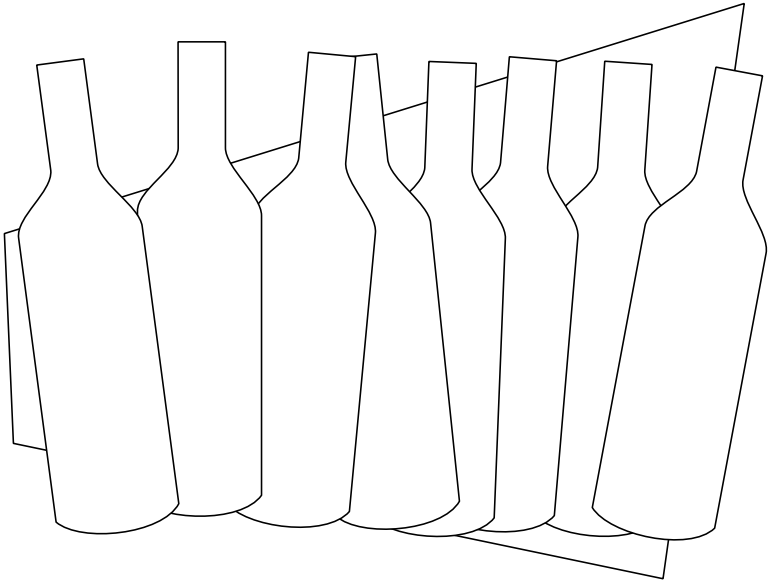
[The command-line interpreter does NOT distinguish between upper and lower case characters in the command name.]

TEN GREEN BOTTLES

True to its title, if you are using a colour TV, you will see 10 green bottle-shapes fined up against a black background (programmed in fine 60). The object of the game is quite simply to knock the bottles off the wall in the shortest possible time.

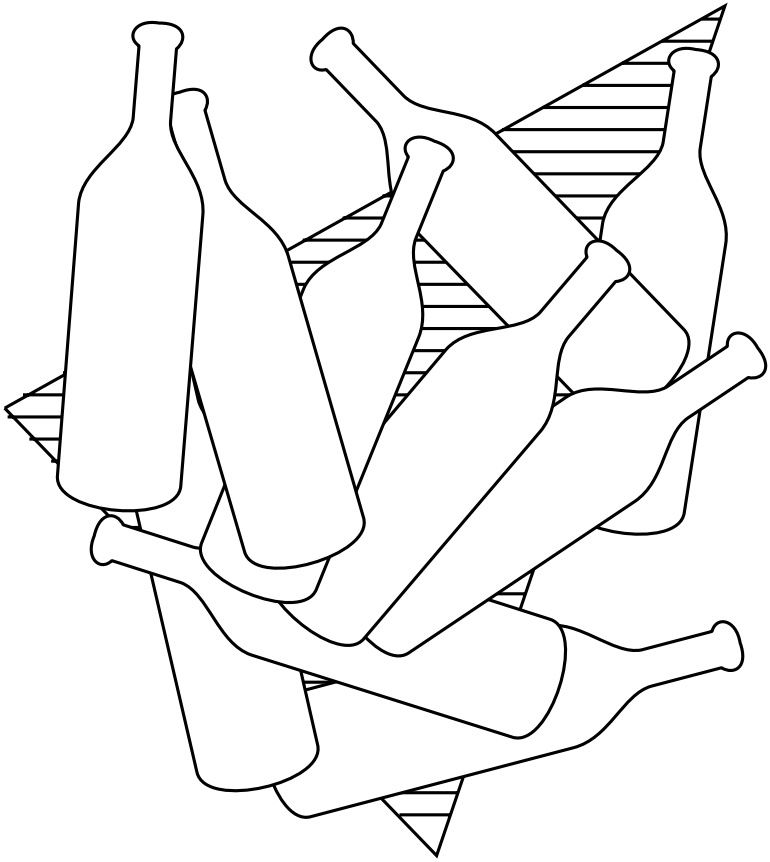
To make matters more difficult the catapult you fire moves constantly across the screen. When you want to fire it press the 'I' key.

When you score a hit you will hear a satisfying sound (programmed in fine 390); at the end of the game the time you took is displayed.



```
  10 REM 10 GREEN BOTTLES
  20
  30 MODE 4
  40 VDU 19,1,2,0,0,0,0
  50 VDU 23;8202;0;0;0;
  60 VDU 23,224,&18,&18,&18,&3C,&3C,&3C
,&3C,&3C
  70 VDU 23,225,&00,&00,&00,&FF,&FF,&00
,&00,&00
  80 VDU 23,226,&3C,&3C,&3C,&FF,&FF,&FF
,&FF,&FF
  90
 100 P=0
 110 H=0
 120 T=0
 130 E=1
 140
 150 CLS
 160 PRINT:PRINT:PRINT:PRINT
 170 DIM A$(10)
 180 FOR X=11 TO 20
 190 PRINT TAB(X,5) CHR$(224)
 200 NEXT
 210
 220 PRINT
 230 FOR X=1 TO 20
 240 PRINT TAB(X,6) CHR$(225)
 250 NEXT
 260
 270 PRINT TAB(P,20);" "
 280 P=P+E
 290 IF P=0 OR P=28 THEN E=-E
 300 PRINT TAB(P,20) CHR$(226)
 310 IF INKEY$(0)="1" THEN 370
 320 *FX 15,0
 330 IF INKEY$(0)<>"" THEN 330
 340 T=T+1
 350 GOTO 270
 360
```

```
370 IF ?FNS(P,5,0)<>&18 THEN 340
380 PRINT TAB(P,5)"O"
390 SOUND 1,-15,0,20
400 A$(INT(P/3)+1)="1"
410 H=H+1
420 IF H<10 THEN GOTO 330
430 PRINT TAB(0,5);"TIME ";T
440 VDU 20
450 END
460
470 DEF FNS(X,Y,N)=HIMEM+(Y*40+X)*8+N
```



CLIFF GOLF

David the golfer has played a rather bad first shot — he has ended up at the top of a cliff far from the green. See if you can guide the ball to the green by suggesting how hard David should hit the ball.

The path of the ball is based on an x^ Y equation, so that it is quite difficult to estimate the strength of shot required. Most of this program is concerned with drawing the cliff and the ground. The program shows triangle drawing and the use of POINT.

```
 10 REM Cliff Golf....
 20
 30 MODE 1
 40 VDU 19,1,2,0,0,0
 50 GCOL 0,1
 60
 70 MOVE 70,0
 80 PLOT 85,100,980
 90 DRAW 0,980
100 PLOT 85,0,0
110 MOVE 1280,0
120 PLOT 85,1280,70
130 DRAW 65,70
140 PLOT 85,65,0
150
160 GCOL 0,2
170 MOVE 380+RND(900),40
180 PLOT 0,0,30
190 PLOT 0,30,0
200 PLOT 81,0,-30
210 PLOT 81,-30,0
220 PLOT 81,0,30
230 MOVE 70,1000
240 GCOL 0,3
250
```

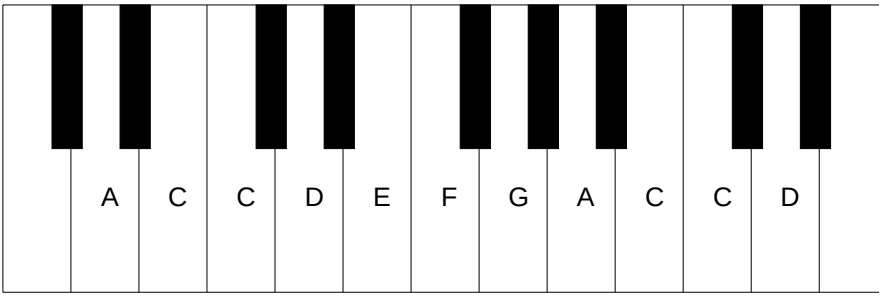
5 Music Theory for Micros

If you stop to consider the matter you will realise that literally every subject of any complexity inevitably acquires its own specialist vocabulary. Music is no exception to this general rule for just as the pioneers discovered that it was necessary to develop languages like BASIC and FORTH to facilitate the use and development of computer technology, so the world of music also has a special language all of its own. Since the scope of music only encompasses pitched notes and rhythm, however, its structure is considerably simpler than that of any computer language. For this reason I believe that most newcomers to music theory who have had some general computer experience will pick up the rudiments of music theory quite easily.

I have felt it necessary to digress somewhat and emphasize this point because of the experience of music teaching many of you will doubtlessly have had at school. For me, school music lessons comprised of note learning, primarily concerned with absorbing key signatures, composers' names and other pieces of musical gobbledygook. In this chapter I do not intend to bombard you with meaningless mnemonics designed to make it easier for you to remember the notes of the treble clef. This type of knowledge can be acquired quite naturally through growing familiarity with music itself. Instead, I intend to provide you with a vocabulary which will allow you to understand the more musically technical sections of this book and, show you how easy it is to use sheet music as reference material.

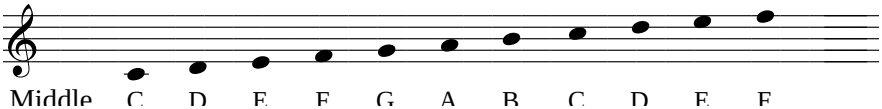
PITCH NOTATION

In order to make it possible for us to speak about individual note pitches we must first start off by giving each note a name. The accepted method for doing this is to call them after the letters in the alphabet between A and G. If you look at a piano keyboard, you will see that the notes are found at the positions shown below:



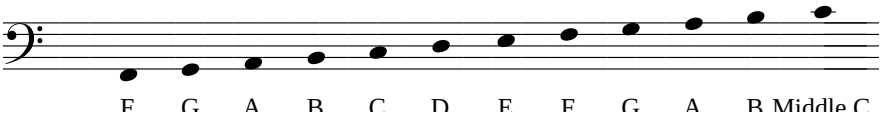
On the BBC Micro we have pitch values such that the A note above Middle C=89, 8=97, C=161, D=109, E=117, F=121, G= 129 and, for the A note one octave above the first, A=137. As you can see, this method of notation has its drawbacks. All the As, in whatever octave, have the same name. Bear in mind that a note which is one octave above another has double that note's frequency.

A better method for illustrating pitch is to abandon numbers and letters altogether and draw them on a musical stave, as is illustrated below:



In this case there is no ambiguity in discriminating between notes of different octaves. In addition, the concept of higher notes being situated higher up the stave is both extremely helpful and logical. A stave is, of course, the name given to the standard arrangement of five lines!

The symbol at the beginning of the stave is called a Treble Clef. This clef is used for the treble register of a piano and all instruments that play in that range, such as flutes, oboes, guitars etc. Bass instruments have their own clef, unsurprisingly called the Bass Clef. This is used by instruments such as trombones, bassoons and double basses. It looks like this:



A piano covers such a large range that it uses both clefs, and we can correlate piano keyboard and notation as follows, with each white note having its own fine or space. BBC Micro pitch values are shown also.


Personal Computer World
CENTURY SOFTWARE
PCW
GAMES COLLECTION
FOR THE
BBC MICRO
DIGITALLY REMASTERED EDITION

GOLDEN FLEECE

By Simon Bryan

Golden Fleece is an interesting adventure game with the added bonus of exciting graphics at each location. The object of your quest is to find the Golden Fleece, which is at one of the 18 locations. You may type in one or two word commands: the commands that the program understands are GO, DIG, GET, DROP, PAY, TAKE, CUT, ROW, KILL, LIGHT, OPEN and INV (inventory - to give you a list of the objects that you are carrying). Directions may be entered in any of three different ways; for example to go north you can type GO NORTH, NORTH or just N.

The which the program is written makes it easy to extend. Each verb is dealt with by its own procedure, so to add an extra verb all you have to do is add a line to check for it, at line 535 for example, and then add a procedure to deal with it.

Program Description

50 Change mode.
60 Change colour 1 to yellow and turn off the cursor.
70-90 Redefine the # $ & % and ' characters.
100-260 Print the Introduction.
270 Change colour 1 to flashing.
280 Wait for a key to be pressed and clear the screen.
300 Change colour 0 (the background) to blue.
310 Change colour 1 to white.
320 Set up the arrays.
330 The start of the firs loop.
340 Set up the variables for a new game.
350 The start of the game loop.
360 Draw the picture for the room that you are in.
370 See if the game has ended.
380 Print a list of the objects in the room.
390 See if Gandalf will appear.

400 Get the command string.
410-430 See if a direction has been typed in.
440-530 Check the fist of verbs and act accordingly.
540 See if the command was recognized.
550 Return to the main game loop.
560-590 See if another game is wanted and act accordingly.
610-650 PROCobject - print a list of objects in the room.
610-100 PROCmove - deal with a movement
120-190 PROCinv - print a fist of the objects that the player carrying.
810-850 PROCdig - deal with the word "DIG".
810-940 PROCget - add an object to the player's inventory.
960-1030 PROCdrop - leave an object in the room.
1050-1080 PROCdelay - delay for a given time unless a key pressed.
1100-1140 PROCcut - deal with the verb "CUT"
1160-1210 PROCpay - deal with the verb "PAY"
1230-1280 PROCrow - deal with the verb "ROW'
1300-1320 PROCkill - deal with the verb "KILL"
1340-1310 PROClight - deal with the verb "LIGHT"
1390-1440 PROCopen - deal with the verb "OPEN"
1450-1520 PROCgandalf - print Gandalfs arrival.
1540-1560 PROCprint - print a given string with a short delay between each letter.
1580-1640 PROCget-command - input a command string split it into two parts - VERB$ and NOUN$.
1660-1680 FNinstr - perform the INSTR command and avoid the bug in this command found in BASIC 1
1100-1110 PROC setup - set up the variables for a new game
1190-1810 PROCpicture - print the picture of a given room
1890-1910 Data for the arrays.
1930-2450 Data for the rooms.

```
10  REM **********************
20  REM * The Golden fleece *
30  REM *    By Simon Bryan *
```

9
Characters

Electron users are able to define their own graphics characters, either to produce special alphabets and symbols or, more often, to use as the basis of graphics displays in games. Designing a single character is easy, and is explained fully in Chapter 20 of the Electron's User Guide. Designing characters is fun, but you need lots of squared paper on which to work out the designs and you may need to make several attempts before you make each design perfect. Even then, the character may not be quite what you intended when you finally see it displayed in its proper size on the screen.

This program provides a large-scale display on the screen on which you draw your design. You use the editing keys, the 'arrow' keys and COPY key of the Electron, for this. The design is easily altered, too. You can change it a little at a time until it isjust right. While you are building it up as a large-scale version in the design area, you see it appearing at the bottom of the screen in its proper size. This is the size it will be when you use it in your own programs. As you work out each design, you will be able to gauge how effectivc it will be in use. When the design is ready, the program calculates all the values that are required for the VDU statement which will define the new character.

The program helps you design more thanjust a single character. To produce larger and more elaborate graphics, we often want to compose a design from two or more characters, placed side by side. As shown in Fig. 9.1, designs made from two or more characters offer much more scope for inventiveness than those made from a single character. The program provides an area on the screen where you can work with up td nine characters at once.

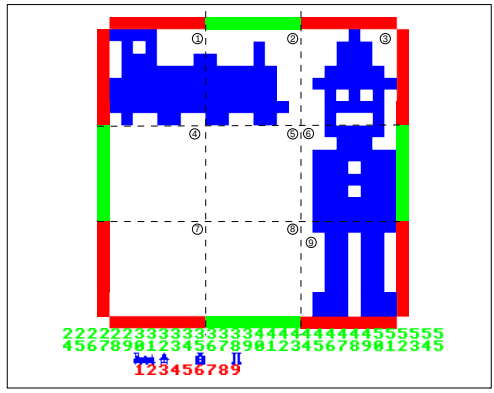


Fig. 9.1. The screen display for CHARACTERS.The dashed lines and encircled numbers are to mark out the individual characters; they do not appear on the screen.

Using the program

As soon as you run the program the display appears as in Fig. 9.1. At this stage the design area, enclosed by the red and green borders is completely blank. The borders are coloured red and green to help you see exactly where the area is sub-divided into three rows ofthree large squares. The dashed lines shown in Fig. 9.1 pick out these squares, but the dashed lines do not appear on the screen.

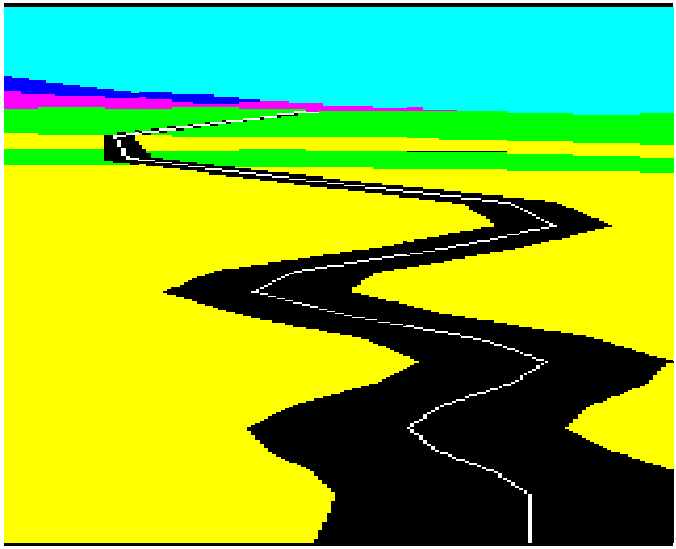
Below the design area is a row of green numbers. It looks like rmo rows of numbers but, if you read domn each column, you can see that these are the numbers '24', '25', '26' and so on, up to '55'. These numbers are to be thought ofas '224', '225', '226', up to '255'. The initial '2' has been left out ofeach numb::r to save a line on the screen. This range of numbers (224 to 255) covers the 32 ASCII codes allocated for user-defined graphics in the Electron. The program allows you to define characters for any or all of these codes. Eventually, the characters you design will appear in a row below their respective numbers. You can see how this will Iook in Fig. 9.1.

The design area allows nine characters out of the available 32 to be designed at one time. When you first run the program, the message 'Which number? (224-247)' will be showing at the bottom of the

13

Get that bird!

If your kids are like the one that has the contract to hike mud into our house, they will love a certain cartoon character that travels at a great rate of knots along a road, always eluding the foxy type who is trying to catch him. In this computer game, they are trying to catch the bird. We suppose that a charge of dynamite has been placed in the road and as soon as the bird appears, the player has to slap any key to detonate the charge. The problem is of course that the bird travels very fast indeed. Actually, I have included skill levels from 0



to 9, and although anyone can win at the lowest level, quite frankly, I think that level 9 is completely impossible - which is to admit that I at least have never won at that level!

The background to the picture comprises sky and hills and contains a random element so that every game is slightly different. When you are tired of playing the game, you can try experimenting with the numeric values of line 90 in order to see the different types of terrain that will occur. What I have done is to divide the screen across into 12 sections and then, starting at the top of the picture, to create random values varying slightly as we move across. These are put into one row of the landscape array, L%(X,Y). Subsequent rows have a smaller deviation, so that the hills gently flatten into undulating land.

The road is always the same and is built around a sine wave decaying as it nears the bottom of the screen. In addition, choosing a step size of -43 (lines 210 and 250) make it impossible for the road to follow the wave closely, thus appearing asymmetrical.

Lines 280 to 310 define four separate characters. These are four round shapes of increasing size, to represent the fleeting bird as it rushes headlong down the picture. In run-time, there is no opportunity to study the bird, and the general impression of something rushing down the road towards the viewer is quite sufficient. If desired however, the last two shapes might be redesigned as a cartoon character.

The game loop is contained in the REPEAT-UNTIL loop of lines 350 to 470 and will repeat forever until the player is successful. There is an indefinite pause in line 360 and then line 370 starts a cackling sound that accompanies the bird. The *FX15 command of line 360 has flushed the input buffer so that the user cannot slap a key before this instant. Line 390 closely mirrors the road construction, so the path of the bird is down the road centre, with lines 400 to 430 choosing a character size to suit the position down the screen.

Line 440 looks for a user input (no RETURN necessary), while line 450 checks if the bird has reached the bottom of the screen. The difficulty level affects the height up the screen that represents safety for the bird, with zero equal to the very bottom. This is checked on line 460. If a key has been hit and the bird is still above the safety level, the program jumps to line 510, where an explosive sound is made and a flashing explosion appears at an appropriate spot in the road.

Of the two procedures, PROCW provides a delay of three one-hundredths of a second for the reason outlined in the Bicycle Wheel program - the image of the spot or bird must be allowed to form before being wiped. A value less than three may make the bird totally invisible, while a larger value will make the game a trifle too easy. Try it and see. PROCPAINT will draw and fill any four-sided

figure by utilising two triangles. The colour is specified before the procedure is called. Notice that we have a number of variables declared as LOCAL; this is a useful provision, because not only may we use the same variable labels elsewhere, but also there is a saving in memory space, which sometimes is important.

Variables

L%(X,Y)	Lanscape heights
LVL%	Diffiuclty level
Z%	Counter for row in landscape array
X%	Position along the row; also horizontal position
F%	Field counter
C1%	Field colour
Y%	Road sections, or vertical position
K	Timer for starting run; also checking for key-pressed condition
Q%	Counter for decreasing amplitude of explosion (decay)

```
  10DIML%(7,11):*FX9,0
  20MODE7:PRINTTAB(5,5);"GET THAT BIRD!"''
  30PRINT"Difficulty level 0-9?"
  40REPEAT:LVL%=GET-48:UNTIL LVL%>=0 AND LVL%<=9
  50
  60REM - Draw background
  70
  80MODE2:FOR Z%=1TO6
  90Y%=RND(30)+(900-30*Z%):L%(Z%,1)=Y%
 100FOR X%=2 TO 11:IF RND(1)>.5 GOTO120
 110Y%=Y%+RND(20)/Z%
 120Y%=Y%-RND(20)/Z%
 130L%(Z%,X%)=Y%:NEXT:NEXT:GCOL0,134:CLG
 140FORF%=1TO6:READC1%:GCOL0,C1%:PROCpaint(F%):NEXTF%
 150ENVELOPE1,1,-26,-36,-45,255,255,255,127,0,0,-127,126,0
 160DATA4,5,2,3,2,3
 170
 180REM - Draw road
 190
 200VDU29,600;0;18,0,5:MOVE0,820:PLOT0,0,0:GCOL0,0
 210FOR Y%=820 TO 0 STEP-43
 220X%=SINY%*Y%*.6
 230PLOT85,(X%+820-Y%),Y%:PLOT81,-820+Y%,0
 240NEXT:GCOL0,7:MOVE0,820
 250FOR Y%=820 TO 0 STEP-43
 260DRAWSINY%*Y%*.6+(820-Y%)*.5,Y%:NEXT
 270GCOL4,0:*FX11,0
 280VDU23,224,192,192,0,0,0,0,0,0
 290VDU23,225,192,192,192,192,0,0,0,0
```

```
300VDU23,226,224,224,224,224,224,224,224,0
310VDU23,227,28,62,255,255,255,255,62,28
320
340
350REPEAT
360K=TIME:REPEAT:UNTIL TIME>=K+RND(1000)+300:*FX15,0
370SOUND1,1,255,255
380Y%=820:*FX12,0
390X%=SINY%*Y%*.6+(820-Y%)*.5:MOVEX%,Y%:*FX15,1
400IFY%<=200PRINTCHR$227:PROCW:PRINTCHR$227:GOTO440
410IFY%<=400PRINTCHR$226:PROCW:PRINTCHR$226:GOTO440
420IFY%<=600PRINTCHR$225:PROCW:PRINTCHR$225:GOTO440
430PRINTCHR$224:PROCW:PRINTCHR$224
440K=INKEY(0)
450Y%=Y%-43:IFY%>=0 AND K=-1 GOTO390
460IFLVL%*50<Y%GOTO510
465*FX15,0
470UNTIL 0
480
490REM - Got him!
500
510VDU29,X%+600;Y%+43;
520FOR Z%=1 TO 10:MOVE0,0
530MOVERND(200)-100,RND(200)
540PLOT81,RND(200)-100,RND(200)
550NEXT:*FX15,0
560FOR Q%=-160TO0:SOUND0,Q%/10,16,1:NEXT
570G$=INKEY$(400):MODE7
580PRINTTAB(5,5);"YOU GOT HIM!"
590G$=INKEY$(500):RESTORE:GOTO80
600
610DEFPROCW:K=INKEY(3):MOVEX%,Y%:ENDPROC
620
630DEFPROCpaint(S%):LOCALLZ%,X%,X1%,A%,B%,C%,D%
640FOR Z%=1 TO 10:X%=(Z%-1)*128:X1%=Z%*128
650A%=L%(S%+1,Z%):B%=L%(S%+1,Z%+1)
660C%=L%(S%,Z%):D%=L%(S%,Z%+1)
670MOVE X%,A%:MOVE X%,C%
680PLOT85,X1%,B%:PLOT85,X1%,D%
690NEXT
700ENDPROC
```

Hexagons

Britain doesn't seem to have drive-in cinemas. In winter it's so cold that nobody would dream of sitting outside in the car for several hours; in summer the light lasts so long in the evening that nothing would show on the screen; and it's almost always too wet anyway.

I used to enjoy going to the drive-in in Australia. It's cheaper than the normal cinema, the atmosphere is much less stuffy, you can dress as you please (not that I don't normally, but my friends feel obliged to dress up for the indoor cinema), and you see two full length feature films rather than one.

But the best thing about the drive-ins, or some of them anyway, is the interval. After they've finished showing those dreadful slide-and-monologue adverts (which you ignore by getting something from the cafeteria), they show the most delightful, relaxing, soothing display I've ever seen outside nature. Everyone else is busy queuing, eating, talking, or whatever else one does in the interval; but I sit there hypnotised, my eyes glued to the screen, until the second film starts.

What, you demand, is this captivating display? It's very simple, but none the less beautiful for it. They project coloured light through bobbly glass (the sort bathroom windows are made of) and very, very slowly tum the wheel holding the colour filter. Another colour gradually starts to seep across the screen - but not just coming in from one side, as you might expect; the refraction effects of the glass scatter the incoming colour across the screen, bobble by bobble. The effect is superb.

It has long been my intention to acquire a slide projector, a bit of bubbly glass, an electric motor geared way down, and a wheel holding various filters, so that I could repeat the effect in my own home. I haven't managed it yet, but I have acquired a computer.

This program uses a repeating pattern of hexagons and diamonds to simulate the glass. Its regularity is rather unrealistic, but you try doing something like this without regularity! Each

basic unit (one hexagon and one diamond) is made up of 596 mode 5 pixels, excluding the lines defining the shapes. The program simulates the seeping light by making 20 passes through each unit, filling in a few more pixels each time.

Unfortunately, the decision as to which pixels should be filled in on which pass is almost impossible to program. The incoming colour must satisfy lots of rules which are not very easy to specify. It must, for instance, start by affecting the far side of each unit, and leave the near side until quite late. While it is theoretically possible to program all such constraints, and then let the computer randomly choose pixels which satisfy them, there are two reasons for not doing it this way. First, it would take the programmer too long to appreciate and specify all the constraints; and second, it would take the computer too long to do it would spend more time discarding unsuitable pixels than it would filling in the new colour.

So I have decided for myself how many pixels should be affected in each pass, and even which ones they are. That information must be passed to the program as data. I apologise for the quantity of it, but assure you that the effect is worth the time you will spend typing it in and proof reading it. (Of course if you have the cassette of the programs, there's not a lot of effort involved.)

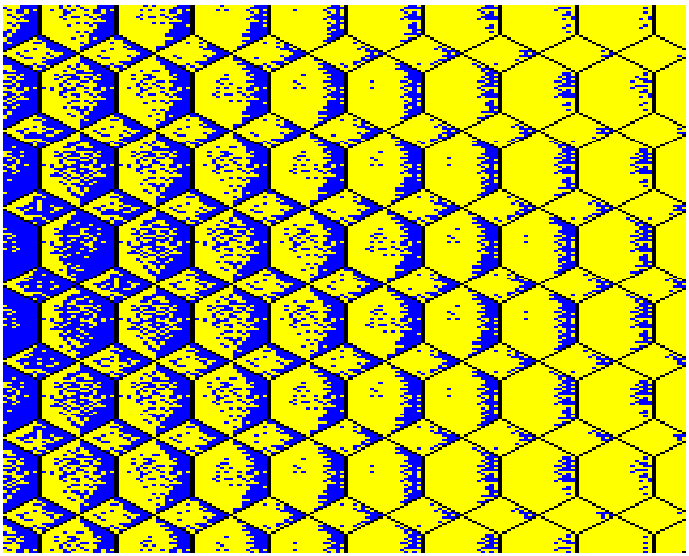
If you have a Model A micro, the quantity of data causes another problem: there isn't room in memory for both the data and the program. I have taken advantage of this situation to demonstrate a feature which even Model B programmers might need now and then. There are two separate programs. The first reads the data, stores it in compact form in an area of memory above the top of either program, and calls the next program. The second program uses the data stored by the first.

How to use the programs

Type in the first program and save it on tape (or disc, if you have one). Type in the second program and save it. You don't have to use the names that I've used for the files, but you must be sure that the name you use in the CHAIN command in the first program is the same as the name you use for the file the second program is SAVEd in. If you have a Model A, don't type the comments in the second program - they take up too much space.

Rewind the tape to a spot before the first program, LOAD it and RUN it (or simply CHAIN it). It will search for the second program on the tape when it is ready.

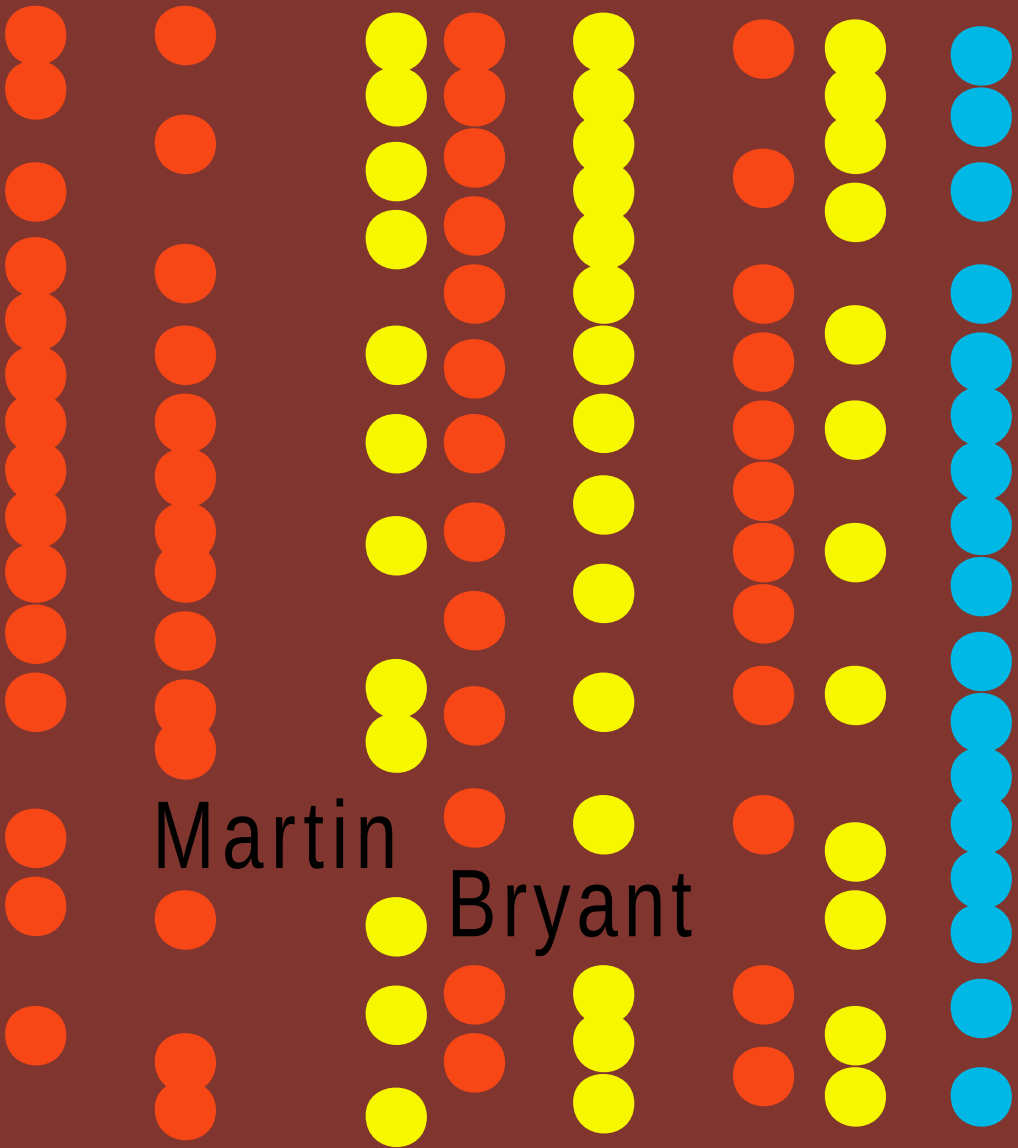
The program will run until broken into with ESCAPE or BREAK. It takes about 20 minutes to complete a full cycle. I see it as fulfilling the same sort of function as a goldfish bowl - something soothing to look at now and then, rather than something to concentrate on.



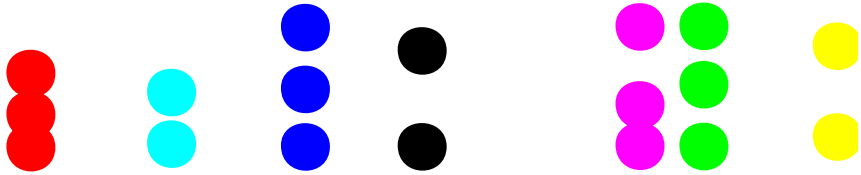
Hexagons, as a darker colour diffuses in from the left

DIGITALLY REMASTERED EDITION

The Second Book of Listings



Martin Bryant



Ricochet Golf

Rules

The rules are as for normal golf; ie hit your ball into the hole using as few shots as possible.

The edges of this golf course, however, are elastic and so the ball bounces off anything that it hits.

Up to nine players can play at once, each taking it in turn to complete the current hole.

Display

The display shows the current hole, its par rating and the current player's name, along with his shot number.

The ball is shown with a fine near it, to show the direction of aim.

When all players have completed the hole, the par ratings for each player are shown on a scoreboard.

Operation

To aim the ball the 'cue' near the ball can be rotated with the keys:
'Z' - rotate cue clockwise
'X' - rotate cue anti-clockwise

To hit the ball press a number key from '1' to '9'. The weakest strength hit is '1' and the strongest (longest) hit a '9'.

Because different display equipment shows different colours better, a facility has been provided to change the foreground and background colours easily! The colours may be moved one at a time through the eight possible colours on the BBC micro.
To advance the foreground colour press 'F'
To advance the background colour press 'B'.
(Note that when the foreground and background colours selected are the same, the hole will 'disappear' until you change one of the colours.) You could, perhaps at a certain stage of a party, invite people to play blind ricochet golf!

Program

The program reads the hole shapes from the data statements at the end of the program. The first number is the par value for the hole, followed by the X, Y coordinates of the apexes, and finally the hole and tee coordinates.

A negative apex X-coordinate signifies an absolute move to the current coordinate pair. A positive apex X-coordinate signifies an absolute draw to the current coordinate pair. The final apex coordinates are specified with a negative Y-coordinate. The hole coordinates specify the centre of the drawn hole. The tee is specified by a lower-X-coordinate, an upper-X-coordinate and a Y-coordinate. The ball is teed off from a random position along the tee line.

Section/Variables	*Function*
Main routine	Initialize data,setup players, main game loop, game over
HCX%	Store hole X-coordinates
HCY%	Store hole Y-coordinates
SC%	Player scores
N$	Player names
BC%	Background colour
FC%	Foreground colour
NP%	Number of players
NH%	Number of holes
TPAR%	Total par
HN%	Current hole number
FN%	Current player number
PROC PLAYHOLE	Play current hole to completion for current player
SH%	Shot number
CH%	Cue angle
LWI%	Last-wall-hit index
K$	Input key
BE	Ball energy
BA	Ball angle
PROC WHOOP	Play 'hole-in-one' fanfare
PROC DELAY	Delay for one second
FN HOLED	Check if ball in hole
PROC MOVEBALL	Move ball when hit
BX	Ball X-coordinate
BY	Ball Y-coordinate
BDDX	Saved increment in ball X-coordinate
EDDY	Saved increment in ball Y-coordinate
BDX	Increment in ball X-coordinate
BDY	Increment in ball Y-coordinate
MISS%	Missed-wall flag
PROC SETD	Set X,Y increments for ball movement

PROC PBALL	Print the ball
PROC MOVECLUB	Erase, move and redraw club
A%	Angle change
PROC PCLUB	Print club
PROC PSCORES	Print player's scoresheet
PROC READHOLE	Read hole 'shape' from data tables
PAR%	Par value for current hole
HCI%	Hole coordinate pair index
HX%	Hole X-coordinate
HY%	Hole Y -coordinate
TLX%	Tee lower X-coordinate
TUX%	Tee upper X-coordinate
TY%	Tee Y-coordinate
PROC PHOLE	Print current hole

Suggestions

Construct your own data statements for a collection of different golf courses!

For variety you could change the course to only play nine holes say, but select which nine randomly from the whole list of eighteen holes (or many more if you add your own).

There is a minor infelicity: The message at the top of the screen can be 'after 1 holes'. Make it grammatical!
(My best score: 9 under par)

The Listing

```
   10 *FX4,1
   20 DIMHCX%(99),HCY%(99),SC%(9),N$(9)
   30 BC%=2:FC%=7
   40 REPEAT
   50   RESTORE
   60   MODE7:PRINTTAB(5,1)"Number of players(1-9)
?";
   70   REPEAT NP%=ASCGET$-ASC"0"
   80     UNTILNP%>=1ANDNP%<=9
   90   PRINT;NP%
  100   FORI%=1TONP%:PRINT'"Name of player ";I%;:I
NPUTN$:N$(I%)=LEFT$(N$,15)
  110     NEXT
  120   FORI%=1TONP%:SC%(I%)=0
  130     NEXT
  140   NH%=18:TPAR%=0
  150   FORHN%=1TONH%
  160     PROCREADHOLE
  170     FORPN%=1TONP%
  180       MODE4:VDU23;8202;0;0;0;
  190       PROCPHOLE
```

THE SUPER-USER'S BBC MICRO BOOK

Brian James and Graham Keeler

Chapter 9

FILE HANDLING

9.1 TYPES OF FILE

The tape and disc filing systems on the BBC microcomputer are not restricted merely to storing BASIC programs. It is also possible to create files to contain data. The data could represent numerical information or text, and if required both types can be stored in a single file. This type of file is frequently called a textfile or data file and since, on some other computer systems, textfile is used to mean a file more like the ASCII file, we shall here refer to such files as data files.

It is worth pointing out straight away that the BBC computer operating system makes no intrinsic distinction between different types of file. They are all stored in the filing system, be it tape, disc or ROM, in exactly the same way.

Thus, any file can be loaded into memory (if there is room) by the command

```
*LOAD <filename> (<Load address>)
```

If the file happens to be a BASIC program, and the load address defaults to, or is given as, the current setting of PAGE, then it will be loaded and can be run just as if it had been loaded with the BASIC command LOAD "<filename>" .

Similarly, you can use the command *SAVE to save a BASIC program, if you know the right addresses to use.

What really creates an effective distinction between different types of file is the way that the information in the file is structured. This structuring will be carried out by the commands used to create the information or to store the information in the file. We can in this sense distinguish at least five separate file types

BASIC program
machine code program
binary file
data file
ASCII file

BASIC programs and machine code programs must have the information structured as the appropriate type of program *before* saving to a file.

Data files and ASCII files have their information structured (differently) by the commands which handle the respective types of file.

Binary files can contain any type of information, including the other four file types and also otherwise incomprehensible information such as graphics dumps.

We can deal fairly rapidly with the types of command associated with three of the file types.

BASIC programs

These are normally created by SAVE and loaded by LOAD or loaded and run in a single operation by CHAIN. These three commands are BASIC commands rather than operating system commands, so they are not preceded by a star, and the filename must be in between inverted commas.

Machine code programs

These can only be saved and loaded with the commands *SAVE, *LOAD and *RUN. The last command is equivalent to *LOAD followed by CALL, and plays a role similar to CHAIN for BASIC programs.

Binary files

Any section of memory, such as the area of a graphics display, can be saved and loaded using *SAVE and *LOAD (use of *SAVE and *LOAD are described in detail in Section 10.8.5). Thus to save a graphics picture in Mode 0, 1 or 2, the command would be

```
*SAVE GRDUMP 3000 8000
```

The picture could be loaded back simply by

```
*LOAD GRDUMP
```

The final two filetypes require special commands, and these are dealt with in the following sections.

Exercise 9.1

Enter a very short BASIC program such as

```
10 PRINT "*SAVE test"
20 PRINT "program compteted"
```

Save it to disc using the command

```
*SAVE TESTPG 1900 19FF
```

(assuming that you have a disc interface in your computer, so that &1900 is the normal program start position)

Type NEW to clear the program, then load it back again with

```
*LOAD TESTPG
```

LIST and RUN the program.

Exercise 9.2

Switch into Mode 1 and generate a simple graphics display, by commands such as

```
MODE 1
MOVE 1,1
DRAW 1000,1
DRAW 1000,1000
DRAW 1,1000
DRAW 1,1
DRAW 1000,1000
MOVE 1000,1
DRAW 1,1000
```

Use the commands given above to save the file with *SAVE, clear the screen with CLS, and then restore the display with *LOAD. (Note, however, that an interesting effect occurs if the screen display is scrolled between *SAVEing and *LOADing.)

9.2 HANDLING DATA FILES

9.2.1 Files and buffers

In BASIC, a set of commands is provided to facilitate handling of data files. Most of the commands can be used with cassette, single-user disc or level 2 Econet systems, and even the less common filing systems such as Prestel and IEEE.

One important difference between the handling of, say, a BASIC program file and a data file is that in the former case the saving or loading of a file is required to be carried out as far as possible as a continuous operation in the minimum of time, whereas for a data file these operations may extend over long periods. For example, in a data logging application, it may be necessary to record measured values of parameters in an experiment or industrial process at intervals of minutes or hours over a period of days. Similarly, where a file contains a large amount of data to be processed during execution of a BASIC program, it will be necessary to read the data in at irregular intervals as and when required by the program.

In order to minimize the number of disc operations, which relative to RAM operations are very slow, a section of memory is allocated for transfer purposes. This section is known as a buffer and has a capacity of 256 bytes, equivalent to one sector of the disc. All transfers between the computer and the disc must go through this memory. For writing to a file, the data in the buffer is transferred to the disc only when the buffer is full or at the end of file handling when the file is closed. In reading from a file, the buffer is filled and

Chapter 8 Special effects with characters and strings

In this chapter, we discuss various special effects that can be obtained with characters and strings on your BBC micro, but first, we present a little more information on how characters aire stored and processed inside the machine.

8.1 How characters are stored

A character is stored inside the computer as an integer that occupies 8 bits or one byte. There is an internationally agreed standard set of codes for the commonly used characters. These arks the ASCII codes (American Standard Code for Information Interchange),

The first table in Appendix 7 contains a list of the normal display characters and their ASCII codes. Actually, in MODE 7, a different international set of characters is used, the Teletext characters. This means that, in MODE 7, a few of the characters displayed on the screen are different from the picture on their keys. However, these characters are used fairly infrequently.

Conversion functions: ASC and CHR$

Two special functions are available for converting the first character of a string into its numeric code (ASC) and for converting a numeric code into a single character string (CHR$). The statement

```
PRINT ASC("+"), ASC("A"), ASC("a")
```

will print the numbers

```
         43          65          97
```

whereas the statement

```
PRINT CHR$(38); CHR$(75); CHR$(122)
```

will print the characters

```
&Kz
```

The following program inputs a sequence of 10 ASCII codes and builds up a string containing the 10 corresponding characters.

```
10 chars$ = ""
20 FOR i = 1 TO 10
30   INPUT "Next code ", code
40   chars$ = chars$ + CHR$(code)
50 NEXT i
60 PRINT "These codes make up "; chars$
```

The VDU statement
The VDU statement provides an alternative way of sendigg characters to the display hardware. The word VDU is followed by a list of character codes separated by commas and these codes are sent one by one to the screen. If the codes represent visible characters, then these characters will appear on the screen. Thus the two statements

```
VDU 65, 66, 67, 88, 89, 90
```

and

```
PRINT "ABCXYZ";
```

have exactly the same effect. However, the VDU statement is normally used for sending invisible characters or special control codes to the display hardware. For example, the ASCII codes from 1 to 31 are reserved for special purposes on the BBC computer and if. one of these codes is sent to the display hardware, it is intercepted and handled specially. For example, 8 is the code for 'backspace' and the statement

```
VDU 8, 8, 8
```

will move the cursor back 3 character positions on the current line. Note that the same ei'fect can be obtained with

```
PRINT CHR$(8); CHR$(8); CHR$(8);
```

or with

```
back3$ = CHR$(8)+CHR$(8)+CHR$(8)
```

```
.
.
.
PRINT back3$;
```

The string '**back3$**' contains three invisible control characters that are sent to the display hardware when the string is printed.

In general there are many different ways of sending a sequence of characters (visible and control) to a device. The most appropriate depends on the application. The last method above would be most convenient in a program that frequently required to send three backspaces. Once, the string has been defined and given a name, the name can be included in a PRINT statement wherever it is required.

Various special control codes will be introduced and 'explained as they are required in this and later chapters. Tables of the various codes and a brief description of their effects appear in Appendix 7,

8.2 Coloured text and its uses

Control of the colour of characters is easily effected on the BBC computer from a BASIC program. The particular facilities discussed in this section are MODE, which selects a particular graphics and text mode for the screen, and COLOUR, which selects particular foreground and background colours for the characters. The modes available to you depend on whether you have a 32K or 16K machine. With the 16K (Model A) machine MODES 7 to 4 are available and with the 32K (Model B) machine MODES 7 to 0 are available. The computer normally operates in MODE 7 and you can always return to this mode by pressing BREAK or by typing MODE 7. For example you may be developing a program that runs in MODE 2. After an erroneous run, a listing in MODE 2 may not be particularly readable, and you could type MODE 7 (without a line number) and LIST the program in MODE 7. Alternatively you could add the following two lines at the bottom of every program:

```
300 keypress = GET
310 MODE 7
```

After a program in say MODE 2 has run pressing any key will clear the screen and return to MODE 7. You can also switch to MODE 7 by typing CONTROL-V followed by 7 or you might prefer to define one of the user-defined function keys to switch to mode 7 and list the program. (See Appendix I to see how to do this.)

A summary of the character facilities and the colours normally available in each mode follows. You can see from this that, in general, as the number of characters available on the screen increases the colour options decrease. This is a point we will be examining in much more detail when we deal with graphics. Mode 7 uses the Teletext standard display characters which cannot be changed by the user.

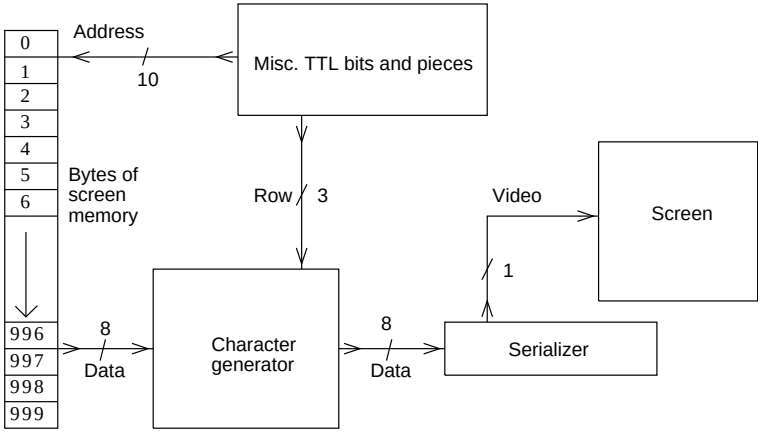
Section one: the 6845 CRTC

The television section of the BBC micro is based around a special chip, the 6845, running in conjunction with the ULA. There are other bits and bobs, but we are not concerned with them for the moment. Both of these chips rival the 6502 as far as complexity is concerned, but the 6845 is considerably easier to use. This chapter describes the hardware used, and how to program the 6845 yourself.

Before discussing the Beeb way of doing things, it is important that you understand how the video section of a typical, old fashioned, micro works. The following account is based on the old PET's video section.

An area of 1000 bytes of memory is used by both the computer and the video circuitry. To the computer this area appears as a normal block of memory, starting at address 32768 and continuing to 33767, assuming the screen format is 25 lines of 40 characters. The video circuitry translates data stored in the memory to the pictures you see on the screen. It does so by accessing each character position of the block in turn, and then displaying the correct character at the correct point on the screen. A description follows the circuit diagram.

Simplified 'PET' VDU circuitry



This circuit is simplified — some of the important points and features have been left out. Each character on the PET screen is made up out of an 8 by 8 matrix, the same as the BBC micro in modes 0 to 6. Thus, there are 64 bits needed to make up each character. These bits are stored in the 'character generator' like this:

ADDRESS	BINARY DATA
0000	00000000
0001	00111100
0002	00100100
0003	00100100
0004	00100100
0005	00100100
0006	00111100
0007	00000000

And so on with the rest of the characters. The character shown above is a 'box' shape. As you can see, eight bytes of storage are required for each character. The type of ROM used for a character generator can hold 2048 bytes, which means that its address bus is 11 bits wide. If you divide 2048 by eight you get 256, which is the total number of displayable characters on the PET screen. 256 characters need eight bits to be represented uniquely. So, the 11 address lines of the character generator are used as follows:

Low order 3 bits — character row (0 to 7)
High order 8 bits — character select (0 to 255)

So to access the data stored in the 5th row of the 45th character, we need to put the following data on the character generator's address lines:

A0 to A2 — 5
A3 to A10 — 45.

You can see the 11 lines going in to the character generator in the diagram. The data bus of the character generator is connected to a serializer, which is a simple chip which accepts eight bits, and then clocks the bits out at a pre-determined rate, one at a time. This chip is typically a 74165.

Thus, to display the fifth row of the 45th character, the above procedure should be carried out, and the required byte will be clocked to the TV by the serializer.

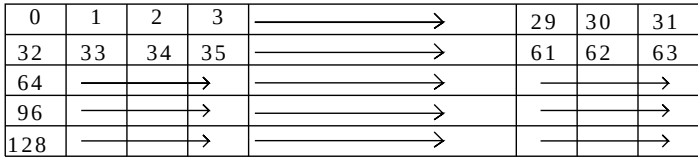
You can also see from the diagram where the eight 'character select' inputs to the character generator come from — they are simply the contents of the memory location currently being accessed in the VDU RAM. The 'row select' signal comes from the TTL bits and pieces. These pieces access the VDU RAM at the right time, with the right row

output to the character generator, eight times, once for each row of each character.

The point of that explanation was to show you how the character generator works. This arrangement is similar to that used in the teletext mode of the BBC computer, except a special character generator is used, the SA5050, and the matrix for each character is much larger, 16 by 16.

The other modes are dot resolution modes. Before discussing these modes, we have to make another comparison, this time with the Atom. The Atom's highest resolution screen is mapped like this, with reference to the start of VDU RAM, which is again 32768:

Atom high resolution screen mapping



etc. . .

7 Editing programs

Introduction

The Electron provides you with a number of very useful facilities for laying out, editing and listing your programs. If you haven‘t done any programming before, here is a brief list of the sort of facilities you will need when typing in programs and making them work:

- Being able to display part or the whole of your program on the screen whenever you want to
- Correcting mistakes, or editing
- Putting comments or notes into the program to help you remember what each part of the program is doing
- Deleting one or more program lines

To start looking at these facilities and how to use them, type in the sample program below which we will use to demonstrate the different facilities.

First, press **BREAK** to clear the screen and reset the computer, then type the following and take care with the punctuation and spaces in the last line.

```
10 PRINT "GIVE ME A NUMBER BETWEEN ONE AND TEN"
20 INPUT X
30 Y=2*X
40 PRINT "TWO TIMES 2;X;" IS ";Y
```

After typing in the above program, type

RUN **RETURN**

When you run this program, the following happens

line 10 **GIVE ME A NUMBER BETWEEN ONE AND TEN** appears on the screen

line 20 A question mark appears on the line below, and the computer waits for you to type in a number which is stored as a variable called X. Type in a number and press **RETURN**

line 30 The computer multiplies X by 2 and stores the result as a

variable called Y

line 40 The following is printed on the screen: **TWO TIMES** (the number you typed in) IS (the result)

If the program won't work properly, or you get an error message, press ESCAPE and type it again – you most likely made a mistake when you typed it in the first time.

Listing the program

When you want to change your program in any way, you will need to display the program (or at least the bit you want) on the screen. To do this, use the BASIC command LIST. Type

LIST RETURN

Your program appears immediately underneath the LIST command on the screen.

If you only want to look at one particular line, say line 40, type

LIST 40 RETURN

Line 40 of your program is displayed on the screen.

To look at a number of consecutive lines, say lines 20 to 40, type

LIST 20,40 RETURN

Lines 20, 30 and 40 appear on the screen.

If you want to see from the beginning of the program up to a particular line, say line 30, type

LIST ,30 RETURN

Lines 10, 20 and 30 appear on the screen.

If you want to see from a particular line to the end of the program, then type

LIST 20, RETURN

Lines 20, 30 and 40 appear on the screen.

Please refer to chapter 25 for a description of the **LISTO** commands. These commands provide you with even more facilities when listing programs.

Editing programs

There are three ways of correcting mistakes in programs you have typed.

One of these you have already met in chapter 5: that is, pressing the **DELETE** key which moves the cursor back along the current line deleting each character as it goes. There is one major drawback to this method – if you have finished typing a line and have pressed **RETURN**, you can't get the cursor to go back to that line by just pressing the **DELETE** key. As we said before, pressing the **DELETE** key only moves the cursor back along the current line, which may not be the one you want to correct.

Another method is to type in the line again, but with the correction. The computer always replaces the old program line with any new version you type in. If the line to be corrected is very short, then this method is fine; but if the line is long or complicated, then use the third method described below.

Editing with the arrow keys and the COPY key
Type

LIST **RETURN**

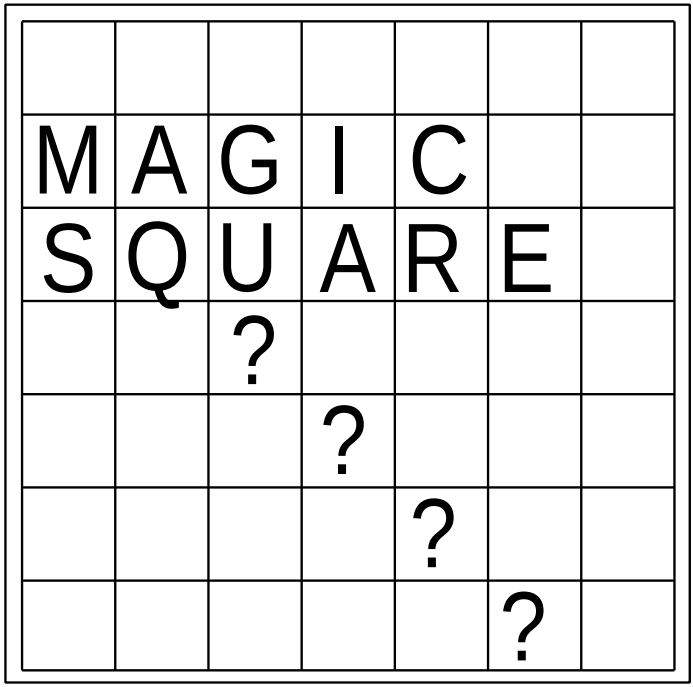
The program appears on the screen, and we are going to use it to try out some editing. The following should now be on your screen:

```
>LIST
  10 PRINT "GIVE ME A NUMBER BETWEEN ONE A
ND TEN"
  20 INPUT X
  30 Y=2*X
  40 PRINT "TWO TIMES ";X;" IS ";Y
>_
```

MAGIC SQUARE

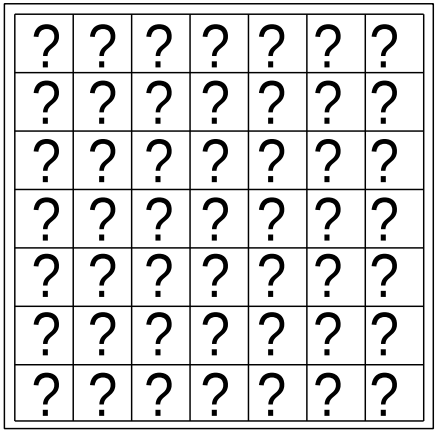
This program will generate four different sizes of magic squares using the de la Loubere method. A 'magic square' is a square of numbers where all the rows, columns, and leading diagonals individually total the same number.

When you run the program, you'll be asked to enter an integer between one and nine. The screen then clears, and a magic square is constructed. The computer then deletes some of the numbers in the magic square and asks you what the mussing numbers are.



```
 20 REM MAGIC SQUARE
 30
 40 MODE 7
 50 DIM M(47,47)
 60 DIM R(12)
 70 DIM Q(12)
 80 INPUT'"NO. OF ROWS/COLUMNS",N
 90 CLS
100 PRINT "MAGIC NO.:"
110 PRINT "ROWS/COLS:  ";N
120 SC=0
130 C1=0
140 C=INT(N/2)+1
150 R=1
160 C1=C1+1
170 M(R,C)=C1
180 PRINT TAB(C*3,R*2+1);C1
190 D=N*N
200 IF C1=D THEN 330
210 IF C1/N<>INT(C1/N) THEN 240
220 R=R+1
230 GOTO 160
240 C=C+1
250 IF C<=N THEN 290
260 C=1
270 R=R-1
280 GOTO 160
290 R=R-1
300 IF R>0 THEN GOTO 160
310 R=N
320 GOTO 160
330 T=0
```

```
 340 FOR I=1 TO N
 350 T=T+M(I,1)
 360 NEXT
 370 PRINT TAB(12,0);T
 380 FOR D=1 TO N+2
 390 C=RND(N)
 400 R=RND(N)
 410 IF M(R,C)>1000 THEN GOTO 390
 420 M(R,C)=M(R,C)+1000
 430 Q(D)=M(R,C)-1000
 440 PRINT TAB(C*3,R*2+1);CHR$(8);CHR$(
129);D;CHR$(135)
 450 NEXT
 460 VDU 28,0,24,39,20
 470 FOR D=1 TO N+2
 480 PRINT'"WHAT IS THE NUMBER AT ";CHR
$(129);D
 490 INPUT R(D)
 500 IF R(D)=Q(D):SC=SC+1
 510 PRINT'CHR$(131);"SCORE:";SC
 520 NEXT
```



DIGITALLY REMASTERED EDITION

PAPERMAC

COMPUTER LIBRARY

WRITING EDUCATIONAL PROGRAMS

FOR THE BBC & ELECTRON

Dave Carlos/Tim Harrison

INCLUDES READY-TO-RUN EDUCATIONAL PROGRAMS

A PRACTICAL GUIDE FOR PARENTS AND TEACHERS

CONTAINS BUILDING BLOCKS FOR CONSTRUCTING NEW PROGRAMS

DESCRIBES PROGRAM WRITING THAT MAKES LEARNING FUN!

'Road sign' test

This program involves a multiple-choice routine that might be of use to you in other situations. It also features a much extended 'PROCdraw_picture' with all kinds of extra facilities. This should give you a few ideas for extending its use in programs you want to write.

The avowed aim is to test the learning of road sign recognition and features twenty different signs, all reproduced on the screen. So that it has a diagnostic value, there is a printout at the end of all the signs answered incorrectly. To give greater variation there are thirty-four possible answers, i.e. some signs are never displayed on screen. Another routine that might be of interest and use elsewhere select the alternative answers. It is called 'PROCread_answers' and incorporates a special checking algorithm which won't allow the same answer to appear twice for the same question.

Figure 10.1

```
10 MODE 5
20 PROCtitle_page
30 PROCinitialise
40 PROCset_order
50 PROCset_screen
60 REPEAT
70   PROCread_answers
```

```
 80   PROCclear_screen
 90   PROCdraw_sign(sign_order(num_signs))
100   PROCset_problem
110   IF input=answer THEN
      PROCright ELSE PROCwrong
120   num_signs=num_signs+1
130 UNTIL num_signs>num_questions
140 MODE 4
150 PROCresults
160 END
```

This is the master program loop and as usual it calls ail the various procedures in the correct order. The workings of each procedure are fully documented below. Line 110 checks the answer given and takes appropriate action, either PROCright or PROCwrong. The REPEAT/UNTIL loop is terminated when the number of signs tested is greater than the number of questions set for this test; this can be changed by altering the value on line 190. PROCresults offers another go, so there is no need to check for this in the main program loop.

Figure 10.2

```
170 DEF PROCinitialise
180 LOCAL answer
190 num_questions=10 : num_answers=34
    : max_questions=20
200 DIM answers(num_questions),
    sign_order(max_questions)
210 DIM question$(4),quest_num(4)
220 num_signs=1
230 num_right=0 : num_wrong=0
240 FOR answer=1 TO num_questions
250   answers(answer)=-1
260 NEXT answer
270 *FX200,1
280 PROCcoff
290 ENDPROC
```

Lines 190, 220 and 230 set up the various numeric constants used in the program. The variable 'max questions' holds the number of pictures in the DATA statements and hence the maximum number of questions that can be asked without restoring the DATA pointer. The arrays are DIMiensioned in this routine and these function as follows:

answers(num_question) holds the wrong answers that are typed in;

sign_order(max_questions) holds the pseudo-random order in which the questions will be displayed;

question$(4) & quest_num(4) are used to hold the possible answers for each of the questions.

The FOR/NEXT loop, lines 240 to 260, is used to initialize the answer array to hold '-1' in each element This value is used to signal a correct answer and this is altered only if a particular answer is incorrect. This is used by PROCprint_answers to signal which ones are to be printed out at the end, The *FX 200,1 ensures that the ESCAPE key has no effect during the running of the program. The cursor is also removed on line 280.

Figure 10.3

```
300 DEF PROCset_screen
310 VDU 26,12
320 PRINT TAB(2,0);
330 PROCdouble("Road Sign Test.")
340 MOVE 180,950 : DRAW 1150,950
350 MOVE 180,955 : DRAW 1150,955
360 VDU 19,0,0,0,0,0
370 VDU 19,1,1,0,0,0
380 VDU 19,2,4,0,0,0
390 VDU 19,3,7,0,0,0
400 ENDPROC
```

This is the screen initialization routine which prints the heading and th• underlines it, lines 320 to 350. The rest of the code simply sets the colours their initial values ready for the first sign.

Figure 10.4

```
410 DEF PROCread_answers
420 LOCAL option,quest_pos,pos,found
430 FOR option=1 TO 4
440   REPEAT
450     found=TRUE
460     quest_pos=RND(num_answers)
470     IF quest_pos=sign_order(num_signs) THEN
        found=FALSE
480     FOR pos=1 TO option
```