Chapter 4 **Animation techniques**

The commonest animation technique that you are likely to use on your BBC micro 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 Word animation and computer-assisted learning

Displaying character information on a screen in a way that is interesting and informative finds applications in many branches of interactive computing. Currently one of the more exotic of these is to replace the conventional electromechanical instrumentation in large complex passenger aircraft with a single animated computer display.

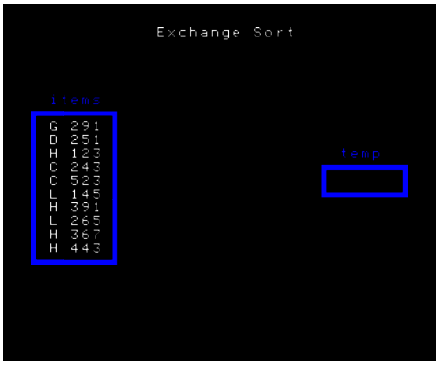
Here as a case study of word animation we look at animating the control flow in a program of reasonable complexity. Explaining to someone how a complicated program works has always been a problem, and in this section, we look at ways of animating a character display to bring a programming technique to life. The technique we shall animate is a simple sort method. Sorting techniques are discussed in Chapter 6, but here we use one of the simplest approaches to sorting a list into order - a simple exchange sort. The program that we wish to animate is presented first. It reads ten items into an array from DATA statements and sorts the items into order. We could think of the data as representing, say, a simple stocklist. Each item is a

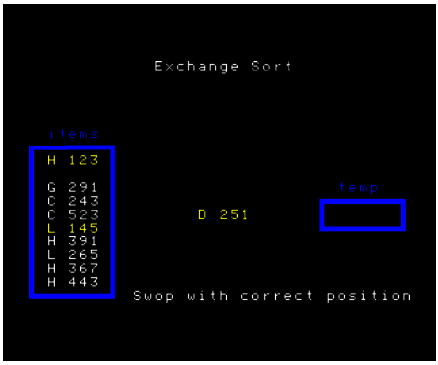
string consisting of a character (a 'department code) followed by a 3-digit integer (a 'stock number'). The items are sorted in the array so that the numbers are In ascending numerical order.

```
100  noofitems=10
110  DIM item$(noofitems)
120  PROCsetuptable
130  PROCexchsort
140  FOR i=1 TO noofitems:PRINT item$(i):NEXT
150  END

200  DEF PROCsetuptable
210    LOCAL i
220      FOR i=1 TO noofitems
230      READ item$(i)
240      NEXT i
250      DATA G 291, D 251, H 123, C 243
260      DATA C 523, L 145, H 391, L 265
270      DATA H 367, H 443
280  ENDPROC

300  DEF PROCexchsort
310    LOCAL i, posnsmallest
320    FOR i = 1 TO noofitems-1
330      PROCfindsmallestentryfrom(i)
340      PROCswop(i, posnsmallest)
350    NEXT i
360  ENDPROC

400  DEF PROCfindsmallestentryfrom(i)
410  LOCAL next
420  posnsmallest = i
430  FOR next = i+1 TO noofitems
440  IF RIGHT$(item$(next),3)
       < RIGHT$(item$(posnsmallest),3)
       THEN posnsmallest=next
450  NEXT next
460  ENDPROC

500  DEF PROCswop(k,1)
510  LOCAL temp$
520    temp$=item$(k):item$(k)=item$(1):item$(1)=temp$
530  ENDPROC
```

What we shall do now is modify the program so that while the contents of the array are being sorted, the contents of the array are also displayed on the screen, and values being moved around in store are also moved around on the screen.

The initial layout of the display that we shall use is illustrated in the first photograph.





The program will run in MODE 7 using Teletext codes to obtain colour effects and Teletext graphics characters to draw the boxes. (For a summary of these codes, see Appendix A. For further information on Teletext consult the User Guide.) Note that a Teletext colour or graphics effect is otained by 'printing' a special character code just before the characters that are to be affected. These special codes appear on the screen as spaces. Their effect lasts only for the line on which they appear. We assume that these codes have been given names at the start of our animation program:

```
10  red$=CHR$(129) : green$=CHR$(130)
20  yellow$=CHR$(131):white$=CHR$(135)
30  blue$=CHR$(132): cyan$=CHR$(134)
40  gb$=CHR$(148) : f1ash$=CHR$(136)
```

where 'gb$' stands for 'graphics blue code'. The other codes will be used for changing the colours of words on the screen, so as to draw attention to them during animation. For example a common technique to animate a scan through a list is to display the list and change the colour of each Item to a highlight colour and back again. The highlight colour then appears to move down the list.

The array 'item$' and the variable 'temp$' together with their contents are initially displayed by PROCsetupdisplay.

```
600  DEF PROCsetupdisplay
610  LOCAL y
620  CLS
630  VDU 23;8202;0;0;0;
640  midx= 16 : leftx=3
650  basey=(24-noofitems) DIV 2
660  tempx=30 : tempy = basey + noofitems DIV 2
```

```
  670  PRINT TAB(leftx,basey-1);blue$;"items"
  680  PRINT TAB(tempx,tempy-2);blue$;"temp"
  690  PRINT TAB(tempx-2,tempy-1);gb$;"h,,,,,,,4";
  700  PROCbars(tempx,tempy)
  710  PRINT TAB(tempx-2,tempy+1);gb$;"*,,,,,,,%"
  720  PRINT TAB(leftx-2,basey);gb$;"h,,,,,,,4";
  730  FOR y=1 TO noofitems
  740  PROCdisplayrec(y,white$)
  750  NEXT y
  760  PRINT TAB(leftx-2,basey+noofitems+1) ;
         gb$; "*,,,,,,,%"
  770  ENDPROC

  800  DEF PROCbars(x,y)
  810    PRINT TAB(x-2,y);gb$;"j";TAB(x+6,y);gb$;"5"
  820  ENDPROC

  840  DEF PROCdisplayrec(r,c$)
  850  PRINT TAB(leftx-2,basey+r) ;
         gb$; "j"; c$; item$(r); gb$; "5"
  860  ENDPROC
```

In order to animate our sort method, we must move a string in our display whenever it is moved in store. To do this, we shall use a procedure PROCmove called in PROCswop.

```
  500  DEF PROCswop(k,l)
  510  LOCAL temp$
  512  LOCAL sk$,sl$
  514    sk$=yellow$+item$(k) : sl$=yellow$+item$(l)
  520    temp$=item$(k):item$(k)=item$(l):item$(l)=temp$
  522    PROCmove(sk$,leftx,basey+k,tempx,tempy)
  524    PROCmove(sl$,leftx,basey+l,leftx,basey+k)
  526    PROCmove(sk$,tempx,tempy,leftx,basey+l)
  530  ENDPROC
```

PROCmove requires 5 parameters. The first pararameter is the string that is to be moved on the screen. We have arranged for the moving string to be highlighted in yellow by including a yellow control code in the string when the procedure is called, the next two are the coordinates of the start position of the string (where it is already displayed) and the final two parameters are the coordinates of the final position of the string.

The most convenient way to arrange for the movement of strings in this type of animation is to establish a highway in the centre of the screen and break all movements down into three stages: horizontal, vertical and horizontal again. The second photograph shows a string in the process of moving up the central highway. In detail, we must:

(1) Move the string horizontally on to the central (vertical) highway.

(2) Move the string up or down the highway to its final vertical position.

(3) Move the string horizontally into its final position.

This approach eliminates the problem of calculating 'trajectories' for the movement between two points and also eliminates the possibility of a moving string wiping out other information that is already on the screen.

Here is the definition of PROCmove together with subsidiary procedures for horizontal and vertical movement. PROCbars is used for restoring the bars at the sides of the array or variable when they have been wiped out by a string moving horizontally into or out of one of the boxes.

```
  900  DEF PROCmove(s$,x1,y1,x2,y2)
  910  LOCAL x,y,xdir,ydir
  920    xdir = SGN(midx-x1)
  930    FOR x=x1 TO midx-xdir STEP xdir
  940    PROCstepx(s$,x,y1,xdir)
  950    NEXT x
  960    PROCbars(x1,y1)
  970    IF y1<>y2 THEN ydir=SGN(y2-y1) :
           FOR y=y1 TO y2-ydir STEP ydir :
             PROCstepy(s$,midx,y,ydir) :
           NEXT y
  980    xdir = SGN(x2-midx)
  990    FOR x = midx TO x2-xdir STEP xdir
 1000      PROCstepx(s$,x,y2,xdir)
 1010    NEXT x
 1020    PROCbars(x2,y2)
 1030  ENDPROC

 1040  DEF PROCstepx(s$,x,y,xdir)
 1050    PRINT TAB(x+xdir-1,y);" ";s$;" ";
 1060    PROCdelay(1)
 1070  ENDPROC

 1080  DEF PROCstepy(s$,x,y,ydir)
 1090    PRINT TAB(x,y);"       ";TAB(x,y+ydir);s$;
 1100    PROCdelay(1)
 1110  ENDPROC

 1120  DEF PROCdelay(d)
 1130  LOCAL t
 1140  t=TIME+d
 1150  REPEAT:UNTIL TIME>t
 1160  ENDPROC
```

The vertical and horizontal movement procedures each require a parameter indicating the direction of the movement. For example in the caae of PROCstepx, the parameter 'xdir' will have the value 1 for movement from left to right and -1 for movement from right to left.

We can further improve the instructive value of the display by adding text explaining what is happening during the sort and by highlighting information in the array in different colours to signify its status as the sort proceeds. For example, once a value has been moved to its correct position, we can highlight it in red. Here are PROCexchsort and PROCfindsmallestentryfrom rewritten to use these facilities.

```
300  DEF PROCexchsort
310    LOCAL i, posnsmallest
315    PROCheading("Exchange Sort")
320    FOR i = 1 TO noofitems-1
325      go=GET:PROCexplain("Find next smallest")
330      PROCfindsmallestentryfrom(i)
335      go=GET:PROCexplain("Swop with correct position"
340      PROCswop(i, posnsmallest)
345      PROCcolourrec(posnsmallest,white$)
346      PROCcolourrec(i,red$)
350    NEXT i
355    go=GET:PROCexplain("Sort completed")
356    PROCcolourrec(noofitems ,red$):go=GET
360  ENDPROC

400  DEF PROCfindsmallestentryfrom(i)
410  LOCAL next
420    posnsmallest = i
425    PROCcolourrec(posnsmallest,yellow$)
430    FOR next = i+1 TO noofitems
435    PROCcolourrec(next,green$):PROCdelay(10)
436    PROCcolourrec(next,white$)
440      IF RIGHT$(item$(next),3)
           < RIGHT$(item$(posnsmallest),3)
         THEN PROCcolourrec (posnsmallest,white$) :
           posnsmallest=next :
           PROCco1ourrec(next,yellow$)
450    NEXT next
460  ENDPROC
```

PROCheading and PROCexplain position a given string in an appropriate place on the screen and PROCcolour simply prints the required colour code before a string at the position specified.

```
1200  DEF PROCheading(s$)
1210    PRINT TAB(13,1);white$;s$
1220  ENDPROC

1230  DEF PROCexplain(s$)
1240  LOCAL spaces
1250    PRINT TAB(leftx+8,basey+noofitems+l);white$;s$;
1260    spaces = 30-leftx-LEN(s$)
1270    PRINT STRING$(spaces," ");
1280  ENDPROC

1290  DEF PROCcolourrec(r,c$)
1300    PRINT TAB(leftx,basey+r);c$
1310  ENDPROC
```

Any sort algorithm can be animated using this technique and the following procedure animates a bubble sort. Again this sort method is explained fully in Chapter 6.

```
1400  DEF PROCbubble
1410  LOCAL i,last
1420    PROCheading("Simple bubble sort")
1430    FOR last = noofitems TO 2 STEP -1
1440      go=GET
1450      PROCcolourrec(1,yellow$)
1460      PROCexplain( "Bubble scan")
1470      FOR i = 2 TO last
1480        PROCco1ourrec(i,yellow$)
1490        IF RIGHT$(item$(i),3) < BIGHT$(item$(i-1),3)
            THEN PROCswop(i,i-1)
            ELSE PROCdelay(10)
1500      PROCcolourrec(i-1,white$)
1510      NEXT i
1520      PROCcolourrec(last,red$)
1530      PROCexplain("Last "+STR$(noofitems-last+1)+
            " now in position")
1540    NEXT last
1550    go=GET
1560    PROCcolourrec(1,red$)
1570    PROCexplain("Sort completed")
1580    go=GET
1590  ENDPROC
```

Exercises

1 Design an animated sequence to illustrate the behaviour of the BASIC statement

```
y=x
```

The sequence should stress the fact that the contents of 'y' are not changed, but are copied into 'x'.

2 Animate the sequence of BASIC statement s for exchange the contents of two variables:

```
temp = x
   x = y
   y = temp
```

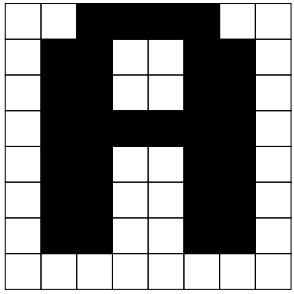
3 When you have read Chapter 6, animate some of the techniques described there; for example, sifting sort, binary search, hash table access and so on.

4.2 User-defined characters

In modes 0, I, 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" is:



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

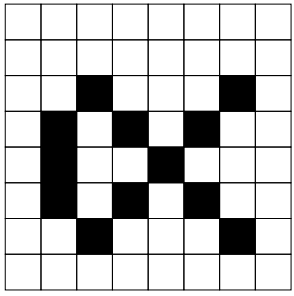
animation. In fact, on lster versions of the operating system (OS 1.2 onwards), it is possible for the user to define shapes for a much greater range of ASCII codes. We shall explain how to do this shortly.

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 2), and each byte can be written as an integer in the range 0 to 255. Thus the above pattern can be described as a list of 8 bytes or a list of 8 integers:

bytes	integers
00000000	0
00000000	0
00100010	34
01010100	84
01001000	72
01010100	84
00100010	34
00000000	0

A byte can also be written in the form of two hexadecimal

digits. Four bits correspond to one hexadecimal digit as described in Appendix 2. Because of this correspondence, it is usually easier to write a pattern of 8 bits in hex than to convert it into an integer:

bytes	hex
00000000	0
00000000	0
00100010	&22
01010100	&54
01001000	&48
01010100	&54
00100010	&22
00000000	0

In order to define our new character shape, we must choose the ASCII code that we are going to use for the character and we must then calculate the sequence of 8 integers (in decimal or hex) that describes its shape. The ASCII code is associated with the required shape by using the VDU 23 command. For example, if we want ASCII character number 224 to appear on the screen as the above shape, we can use

```
10  VDU 23, 224, 0, 0, &22, &54, &48, &54, &22, 0
```

We could equally describe the shape by writing the bytes in decimal :

```
10  VDU 23, 224, 0, 0, 34, 84, 72, 84, 34, 0
```

We can display this character in the centre of the screen in MODE 4 by:

```
20  MODE 4
30  PRINT TAB(20,10); CHR$(224)
```

or by:

```
20  alpha$ =CHR$(224)
30  PRINT TAB(20, 10); alpha$
```

or we can incorporate it in strings:

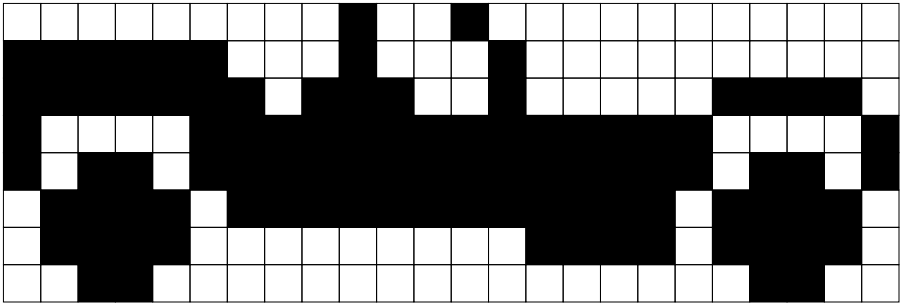
```
20  alpha$ = CHR$(224)
30  particle$ = alpha$ + "—particle"
40  ray$ = alpha$ + "—ray"
50  PRINT ray$; "s consist of a stream of";
      particle$; "s."
```

Here are some further VDU 23 statements for defining the next three letters in the Greek alphabet.

```
20  VDU 23, 225, 0, 0, &1E, &12, &3C, &24, &7C, &40
30  VDU 23, 226, 0, 0, &42, &24, &18, &24, &24, &18
40  VDU 23, 227, &C, &10, &10, &8, &3C, &44, &38, 0
```

Composite character shapes

We can build up bigger objects by defining a number of different character shapes that can be printed together to make up the overall shape of the object. For example, let us define a vintage car for use in MODE 1 or 4. We use a row of three characters for the basic car shape:



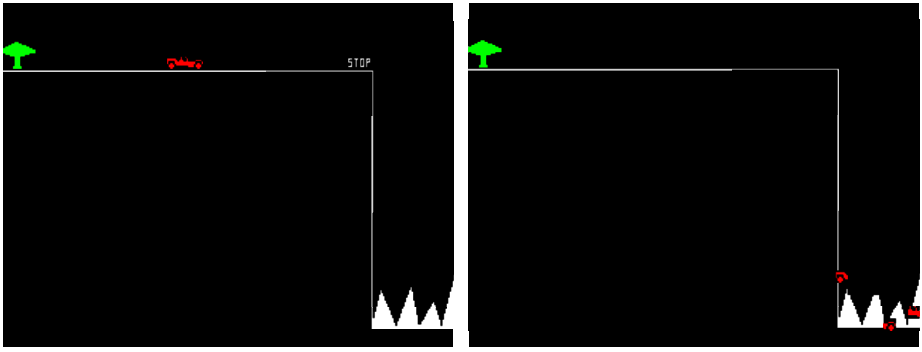
The three characters required are defined by the bit patterns:

```
00000000   01001000   00000000
11111100   01000100   00000000
11111110   11100100   00011110
10000111   11111111   11100001
10110111   11111111   11101101
01111011   11111111   11011110
01111000   00000011   11011110
00110000   00000000   00001100
```

Here is a program that uses these characters together with some other groups of user-defined characters.

```
  10  MODE 1 : VDU 23;8202;0;0;0;
  20  VDU 19,2,2,0,0,0
  30  PROCdefineshapes
  40  PROCbackground
  50  COLOUR 2 : PRINT TAB(1,3)tree$
  60  y=3
  70  COLOUR 1 : PRINT TAB(2,y)car$

  80  FOR x=2 TO 31
  90    SOUND 0,-10,6,1 : SOUND 0,-11,7,1
 100    TIME=0 : REPEAT:UNTIL TIME>10
 110  COLOUR1 : PRINT TAB(x,y)" ";car$
 120  NEXT x

 130  x=32
 140  FOR y=4 TO 22
 150    PRINT TAB(x,y-1)"   "
 160    PRINT TAB(x,y)car$
 170  NEXT y

 180  PRINT TAB(x,22)"   "
 190  VDU 28, 31,25, 39,21
 200  PRINT TAB(RND(7),RND(5));CHR$224;
 210  PRINT TAB(RND(7),RND(5));CHR$225;
 220  PRINT TAB(RND(7),RND(5));CHR$226;
 230  SOUND 0, -10 ,6 ,20
 240  K=GET : MODE 7
 250  END

 260  DEF PROCdefineshapes
 270    VDU 23, 224, 0,&FC,&FE,&87,&B7,&7B,&78,&30
 280    VDU 23, 225, &48,&44,&E4,&FF,&FF,&FF,3,0
 290    VDU 23, 226, 0,0,&1E,&E1,&ED,&DE,&DE,&C
 300    car$=CHR$224 + CHR$225 + CHR$226
 310    VDU 23, 227, 0,&77,&42,&72,&12,&12,&72,0
 320    VDU 23, 228, 0,&77,&55,&57,&54,&54,&74,0
 330    st$=CHR$227 + CHR$228
 340    VDU 23, 240, 0,0,0,0,0,0,&38,&FE
 350    VDU 23, 241, &3,&F,&3F,&FF,&FF,&FF,&F,1
 360    VDU 23, 242, &FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
 370    VDU 23, 243, &C0,&F0,&FC,&FF,&FF,&FF,&F0,&80
 380    VDU 23, 244, &3C,&3C,&3C,&3C,&3C,&3C,&7E,&7E
 390    tree$=CHR$244 +CHR$8 +CHR$8 +CHR$11 +
          CHR$241 +CHR$242 +CHR$243 +
          CHR$8 +CHR$8 +CHR$11 + CHR$
 400  ENDPROC

 410  DEF PROCbackground
 420    PROCrocks
 430    PRINT TAB(29,3)st$
 440  ENDPROC
```

```
450   DEF PROCrocks
460     GCOL 0 ,3
470     MOVE 0,888
480     DRAW 999,888
490     DRAW 999,200
500     DRAW 1020,300
510     PLOT 85,1060,200
520     DRAW 1100,310
530     PLOT 85,1120,200
540     DRAW 1160,270
550     PLOT 85,1180,200
560     DRAW 1220,350
570     PLOT 85,1220,200
580   ENDPROC
```



We have moved the car horizontally and vertically in exactly the same way as we moved words in Section 4.1. There are a number of other interesting features in this program:

(1) The string 'tree$' has been made up of a combination of user-defined characters, backspace characters (CHR$ 8) and 'up' characters (CHR$ 11).

(2) The 'stop' sign consists of two user-defined characters - we can obtain small letters in modes whose normal characters are too large.

(3) The crash is simulated by printing the three separate characters constituting the car at random positions in a text window drawn round the rocks. The text window is created by the VDU 28 statement at line 190.

Designing characters on paper is a tedious process and it is sometimes useful to have a computer program that assists us in the design process. A listing of such a program appeared in our earlier book (The BBC micro book: BASIC, Sound and

Graphics) and this program (with others) is also available on cassette. The use of mult-frame character images was extensively covered in the earlier book and we will not go into it here. However, we shall be using user defined characters in various contexts in the rest of this chapter.

'Exploding' the space for user-defined characters

Under normal circumstances, the user can define up to 32 of his own character shapes for ASCII codes 224 to 255 (&E0 to &FF). The bit patterns defining the shapes of these characters are stored by the operating system at store locations &C00 to &CFF. Actually, codes 128 to 255 all initially refer to this space, four codes corresponding to each character shape. For example, once character 224 has been defined, characters 128, 160 and 192 (&80, &A0 and &C0) all produce the same shape as character 224. When constructing large numbers of frames for large multi-character images, more than 32 new character shapes may need to be defined. The space used by the operating system for storing character shapes can be extended or 'exploded' by using the ccmmand

```
*FX 20, 1
```

After this ccxmnand has been obeyed, all the character codes from 32 up to 255 can be redefined. However, the operating system stores the bit pattern defining most of these shapes at the bottom of the storage area that is normally occupied by the user's BASIC program and the above command must be issued before any program that uses this facility is loaded. The system must then be told to load the program above the space needed for character definitions by changing the value of the system variable PAGE which contains the address of the storage location at which the user's BASIC program is stored. PAGE must be increased by an amount which depends on the character codes that are going to be redefined.

character codes to be defined	increase in PAGE
128-159 (&80-&9F)	no change needed
160-191 (&AO-&BF)	PAGE = PAGE + &100
192-223 (&C0-&DF)	PAGE = PAGE + &200
224-255 (&E0-&FF)	PAGE = PAGE + &300
32-63 (&20-&BF)	PAGE = PAGE + &400
64-95 (&40-&5F)	PAGE = PAGE + &500
96-127 (&60-&9F)	PAGE = PAGE + &600

Exercises

1 Add some more sound effects to the car program, for example a squeal of brakes.

2 Add an extra tree in the middle of the car's route towards the cliff edge. When the car has passed in front of the tree, the tree trunk will need to be redrawn.

3 Arrange for the 'stop' sign to fall down the cliff with the car and break up on the rocks.

4.3 Arcade game animation

The really elaborate commercial arcade games are programmed in assembly code and take anything from two to six man months to write. Features of the computer that are not easily accessible to BASIC programmers are employed. Another important factor is speed. An assembly code program executes more quickly than its equivalent written in BASIC. This is because it is generally more efficient and also when a BASIC program is being executed the interpreter is executed at the same time. Such speed is important in animated games in general but particularly so in games where many animated events are (apparently) taking place simultaneously at different points on the screen.

As long as we are not too ambitious we can write interesting games in BASIC, but we must take care that the techniques we use in our programs are as efficient as possible. In this section, we shall look at two aspects of nuch games - keeping track of the status of a number of moving objects and keeping track of where moving objects are allowed to go (for example, in a Pacman type maze).

Even if we were going to do serious arcade game programming in assembly language, we would need techniques similar to those that we are going to describe, and these techniques are best introduced in BASIC.

Keeping track of moving objects

Arcade game animation of any complexity usually involves two representations of the objects being moved. First of all there is the screen display, which is represented inside the computer by the contents of the screen memory. A code stored in the screen memory indicates the colour of the corresponding pixel on the screen. Such a representation is not convenient for keeping track of the position and status of an object such as a spaceship, which would occupy more than one pixel on the display. We usually have to store separately additional information about an object: for example, coordinates, direction of motion, orientation, etc. It is this information that is repeatedly examined by the program when deciding what action to take next and the number of objects that can be handled in a BASIC arcade game depends on the speed at which this information can be examined and updated. The choice of representation is often critical, making the difference between a slow-moving and uninteresting game, and a fast-moving successful one. We

shall illustrate this point with a simple space invader game.

Moving groups of objects - a fleet of space invaders

There are two approaches to the problem of keeping track of a number of objects. Where objects are scattered about the screen, there is no alternative but to store a list of their coordinates, this list being updated each time the objects are moved. The number of separate objects that can be handled quickly enough in this way in a BASIC program is fairly small.

In the case of a group of objects such as a fleet of 'space invaders' which are moving in unison in a tight formation, it is often more convenient to handle the group as if it were a single object. We can use a single string to represent each row of invaders (or indeed a single string to represent the whole fleet). Here is a simple introductory program that moves a small fleet of 'spaceships' backwards and forwards across the screen, each pass bringing them a step closer to the bottom of the screen.

```
 10  MODE 5
 20  VDU 23,1,0;0;0;0;
 30  VDU 23,224,&18,&18,&18,&3C,&7E,&C3,&7E,&3C
 40  VDU 23,225,0,&24,&7E,&5A,&7E,&7E,&FF,&C3
 50  VDU 23,226,&A5,&FF,&FF,&18,&DB,&99,&FF,0
 60  DIM fleet$(3)
 70  fleet$(0)="   "+CHR$(224 )+" "+CHR$(224)+"   "
 80  fleet$(1)=" "+CHR$(225)+" "+CHR$(225)+" "+
       CHR$(225)+" "+CHR$(225 )+" "
 90  fleet$(2)=fleet$(1)
100  fleet$(3)="  "+CHR$(226)+" "+CHR$(226)+" "+
       CHR$(226)+"  "
110  fleetx%=0: fleety%=0:fleetdir%=1
120  PROCprintfleet
130  REPEAT
140    PROCmovefleet
150  UNTIL fleety%=27
160  END

300  DEF PROCprintfleet
310  LOCAL r%
320    FOR r%=0 TO 3
330      COLOUR r% MOD 2 + 1
340      PRINT TAB(fleetx%,fleety%+r%);fleet$(r%)
350    NEXT r%
360  ENDPROC
```

```
400  DEF PROCmovefleet
410    fleetx%=fleetx%+fleetdir%
420    IF fleetx%>10 OR fleetx%<0 THEN
         fleetdir%=-fleetdir% :
         fleetx%=fleetx%+fleetdir% :
         PRINT TAB(fleetx%,fleety%);"       " :
         fleety%=fleety%+1
430    PROCprintfleet
440  ENDPROC
```

Grouping the spaceships into strings in this way makes the animation considerably more convenient (and faster) than it would be if the coordinates for each ship were stored separately and each ship moved in turn.

An interesting alternative to the use of the COLOUR statement at line 330 would be to include colour control codes in the strings representing the fleet. For example, printing the string:

```
CHR$(17) + CHR$(1)
```

is exactly equivalent to obeying the statement:

```
COLOUR 1
```

(see Appendix 3). Strings like the above could be added to the start of each row of the fleet.

Removing ships from the fleet

There are various types of arcade games that could be developed from the previous program. For example, we shall shortly introduce a 'gunsight' controlled by a user at the keyboard whose aim is to destroy the invading fleet. If one of the invaders is destroyed, then clearly it must be replaced by a space in the string in which it is stored. A further elaboration appears in games of the 'Galaxian' family where spaceships are repeatedly selected from the main fleet to launch an individual attack on the user and his gun. This involves searching through the fleet to find such an attacker, deleting it from the fleet and then keeping a separate record of its subsequent movements.

Changing individual characters in a BASIC string is a fairly cumbersome process. For example, to remove the second invader from the third row of the fleet, we could use

```
fleet$(3) = LEFT$(fleet$(3),4) +" "+ RIGHT$(fleet$(3),4)
```

As well as being cumbersome, this process is slow - the whole string is copied as a result of obeying the above statement. All that really needs to be done is to overwrite one character in the string and we now introduce an alternative method of storing a string that makes changing

an individual character in tho string easier and quicker.

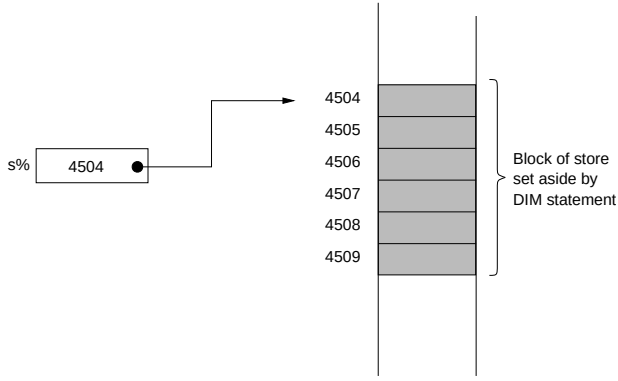
Indirection operators - $ and ?

The BBC computer store consists of a sequence of numbered storage locations or 'words' where each word contains one byte (see Appendix 2). A string simply consists of a sequence of bytes (character codes) stored in consecutive words of computer store. When we store a string in a BASIC string variable, we do not have access to the individual bytes, except by using the MID$ function. We cannot easily change one of the characters in the string without copying the whole string.

An alternative method for storing a string of fixed size is to allocate a block of storage locations in which we can store the character codes of our string. We use an ordinary BASIC variable in which to record the number or 'address' of the storage location at the start of the block. The block of store is allocated by a variant of the DIM statement. For example,

```
DIM s% 5
```

allocates a block of store containing 6 locations and stores the address of the start of the block in the variable 's%'. For instance, the computer might allocate a block of store starting at location 4504

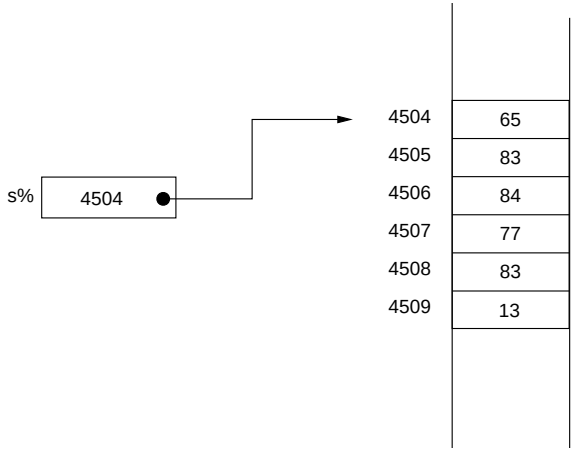


We can now store a string of up to 5 characters in this block by using, for example:

```
$ s% = "ASTMS"
```

The $ operator is called a 'string indirection' operator and it means that the string is to start in the computer word

whose address is held in 's%'. The end of the string is marked by an additional character code 13.



We can print the string by:

```
PRINT $ s%
```

Storing the string in this way allows us to change Individual characters in the string without copying the rest of the string. To do this we can use the indirection operator '?'. The '?' can be used either as a unary operator or as a binary operator. For example,

```
PRINT ? s%
```

would print 65. The '?' causes the computer to refer to the storage location whose address is the value of 's%'. This corresponds to the use of PEEK operations in other BASIC dialects. The same effect could be obtained by

```
PRINT ? 4504
```

but it is better not to rely on the computer allocating the same storage locations every time the program is run. We can change the first character by

```
? s% = ASC("B")
```

which corresponds to the POKE operation in other BASIC dialects. In order to access the third character in the string, we could use

```
PRINT ? (s% + 2)
```

In this case the computer refers to the storage locations whose address is the value of the expression on follow the '?'. However, the '?' can be used bn binary operator and the above is equivalent to

```
PRINT s% ? 2
```

Now we can access the ith code in our string by

```
PRINT s% ? i
```

and change it by, for example

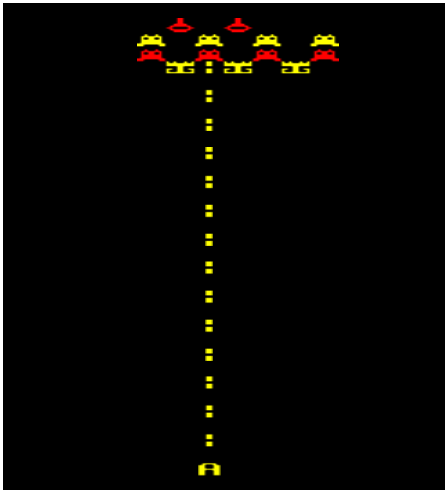
```
s% ? i = 32
```

which sets the ith. character to ASCII code 32 which represents a space.

The next program illustrates various ways of using the above facilities.

Shooting down space invaders

This program is the kernel of a space invaders or Galaxian type program for you to experiment with. The program moves the same fleet of invaders down the screen and the user at the keyboard controls a 'gunsight' (a letter A) that moves left and right at the bottom of the screen. Pressing the space-bar causes a 'laser' to fire at the invading fleet.



As it stands, it is quite easy to shoot down the invaders, but there are many ways in which the program could ae improved. Some of these are suggested as exercises below. There are a number of important points illustrated in this

program and these are explained below.

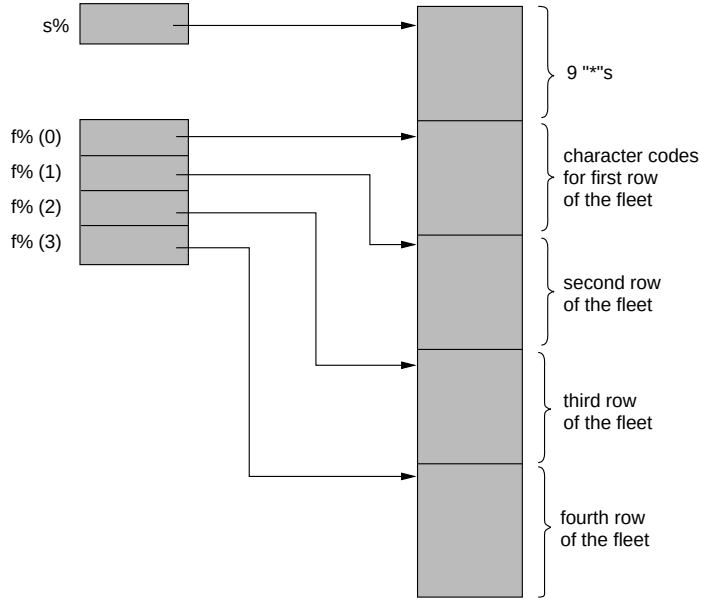
```
 10  MODE 5
 20  VDU 23,1,0;0;0;0;
 30  VDU 23,224,&18,&18,&18,&3C,&7E,&C3,&7E,&3C
 40  VDU 23,225,0,&24,&7E,&5A,&7E,&7E,&FF,&C3
 50  VDU 23,226,&A5,&FF,&FF,&18,&DB,&99,&FF,0
 60  DIM fleet$(3)
 70  fleet$(0)="  "+CHR$(224)+" "+CHR$(224)+"   "
 80  fleet$(1)=" "+CHR$(225)+" "+CHR$(225)+" "+
       CHR$(225)+" "+CHR$(225)+" "
 90  fleet$(2)=fleet$(1)
     100 fleet$(3)="  "+CHR$(226)+" "+CHR$(226)+" "+
       CHR$(226)+"  "
110  DIM f%(3)
120  DIM s% 49
130  $s% = "**********"
140  FOR r%=0 TO 3
150  f%(r%) = s%+(r%+1)*10
160  NEXT
170  PROCinitialise
180  REPEAT
190  PROCgun
200  PROCmovefleet
210  UNTIL fleety%=27 OR invaders=0
220  *FX 12,0
230  MODE 7 : END

300  DEF PROCinitialise
310  *FX 11,5
320  *FX 12,5
330  FOR r%=0 TO 3
340  $f%(r%) = fleet$(r%)
350  NEXT
360  fleetx%=0 : fleety%=0 : fleetdir%=1
370  fdelay%=10 : ftime%=fdelay% : TIME = 0
380  invaders = 13
390  PROCprintfleet
400  gx%=10 : gy%=31 : gun$="A"
410  PRINT TAB(gx%,gy%);gun$;
420  invaders% = 13
430  b$=CHR$(11) + CHR$(11) + "|" + CHR$(8)
440  e$=CHR$(11) + CHR$(11) + " " + CHR$(8)
450  ENDPROC

500  DEF PROCprintfleet
510  LOCAL r%
520  FOR r%=0 TO 3
530  COLOUR r%MOD 2 + 1
540  PRINT TAB(fleetx%, fleety%+r%);$f%(r%)
550  NEXT r%
560  ENDPROC
```

```
  600  DEF PROCmovefleet
  610  IF TIME<ftime% THEN ENDPROC
  620    ftime% = TIME + fdelay%
  630    fleetx%=fleetx%+fleetdir%
  640    IF fleetx%>10 OR fleetx%<0 THEN
           fleetdir%=-fleetdir% :
           fleetx%=fleetx%+fleetdir% :
           PRINT TAB(fleetx%,fleety%);"          " :
           fleety%=fleety%+1
  650    PROCprintfleet
  660  ENDPROC

  700  DEF PROCgun
  710    LOCAL k$, nx%
  720    k$=INKEY$(0)
  730    IF k$=" " THEN PROCfire: ENDPROC
         ELSEIF k$="Z" THEN nx%=gx%-1
         ELSEIF k$="X" THEN nx%=gx%+1 ELSE ENDPROC
  740    IF nx%<0 THEN nx%=0 ELSE IF nx%>18 THEN nx%=18
  750    PRINT TAB(gx%,gy%);" "; TAB(nx%,gy%);gun$;
  760    gx%=nx%
  770    *FX 15,1
  780  ENDPROC

  800  DEF PROCfire
  810  LOCAL r%
  820    PROCtesthit
  830    PROCtrack(r%,b$)
  840    PROCtrack(r%,e$)
  850    PRINT TAB(gx%,r%);" ";
  860    *FX 15 ,1
  870  ENDPROC

  900  DEF PROCtesthit
  910  LOCAL p%, y%
  920    IF gx%<=fleetx% THEN r%=0:ENDPROC
  930    IF gx%>fleetx%+7 THEN r%=0:ENDPROC
  940    p%=f%(3)+gx%-fleetx%+10 : y%=fleety%+4
  950    REPEAT : p%=p%-10 : y%=y%-1
  960    UNTIL ?p%>32
  970    IF ?p%=42 THEN r%=0: ENDPROC
  980    ?p%=32 : invaders=invaders-1 : r%=y%
  990  ENDPROC

 1000  DEF PROCtrack(r%,s$)
 1010  LOCAL y%
 1020    PRINT TAB(gx%,gy%);
 1030    FOR y%= 29 TO r% STEP -2
 1040      PRINT s$;
 1050    NEXT y%
 1060  ENDPROC
```

As before, the array 'fleet$' contains the strings representing the complete fleet, but in this program these strings are copied into a separate block of store for use during animation. The statement:

```
120 DIM s% 49
```

allocates a block of 50 locations and this block is divided up into 5 sub-blocks. The first string (accessed as '$ s%') contains 9 stars and the reason for this is explained shortly. The next four groups of 10 locations each contain one row of the fleet (9 characters plus character code 13). The four addresses of these sub-blocks are stored in the locations of the array f%. Before the attack, the fleet is copied (by PROCinitialise) into the four sub-blocks. (lines 330 to 350). We can picture the representation of the fleet as:



The movement of the fleet is organised in exactly the same way as before. the only difference now is that row 'r%' of the fleet is printed by

```
540   PRINT TAB(fleetx%,fleety%+r%);$f%(r%)
```

which tells the computer to print the string contained in the block of store whose address is contained in f%(r%). One

advantage of storing the fleet in this way is that we can scan rapidly through a sequence of the locations that contain the fleet by using the '?' operator. The way in which we have taken advantage of this in the program is in the check to see whether an invader has been hit when the 'gun' is fired (PROCtesthit).

If the x-coordinate of the gun is within the appropriate range, we must examine the fleet locations in the line of fire and find the invader (if any) that is closest to the gun. PROCtesthit thus calculates an address 10 beyond the location to be checked in the fourth row of the fleet (at line 940). By repeatedly subtracting 10 from the address we examine all the locations of the fleet that are in the line of fire. The REPEAT loop terminates (line 960) if a location is found with a character code greater than 32. Thus it stops when an invader is found, or when it reaches one of the "*"s (code 42) which were put there for that very purpose. We can be sure that the loop will terminate at a "*" even if there are no invaders in the line of fire. Using a block of markers in this way eliminates the need for an additional test to see if we have reached the start of the fleet.

Once PROCtesthit has calculated the range (r%) of the laser shot, the laser effect is created by very rapidly printing a column of dashes up the screen and then equally rapidly deleting them (with spaces). This is done by PROCtrack called twice by PROCfire.

Another interesting aspect of this program is the way in which the movement of the fleet has been slowed down. This has been done in a way that does not create unwanted side effects on the speed of any other objects being moved by the program. If, for example, a delay loop was inserted at line 205 this would not only slow the fleet down, but would make the response to the keys controlling the gunsight very sluggish. A much better approach is to keep a record of the next time (ftime%) that the invaders are to be moved and if this time has not yet ben reached, then PROCmovefleet exits immediately. When the fleet is moved, 'ftime%' is increased by 'fdelay%', the time delay that is required between fleet movements. Changing the value of 'fdelay%' changes the speed of the fleet without having any effect on the behaviour of other objects being moved by the program. Of course we cannot increase the speed of the fleet indefinitely. We are limited by the speed at which the BASIC interpreter can cycle through the main loop in the program.

Finally note the way that the command

```
*FX 15, 1
```

is used after a key has been processed. This command flushes any waiting characters that are queued up in the input buffer and simply avoids a build-up of unprocessed input

characters. The other two *FX commands used are

```
*FX 11, 5
*FX 12, 5
```

which changes the 'auto-repeat' timings of the keys and makes them more responsive.

```
*FX 12, 0
```

changes the 'auto-repeat' behaviour back to normal. (See the User Guide for details if you are not familiar with this facility.)

Exercises

1 As the space invaders program stands, it is quite easy to shoot down the invaders by holding down the fire button. Arrange for a forced delay, or 'reload time' after each firing. The fire button should have no effect during this period.

2 Structure the program so that it repeats the game after each player has finished. Record the time taken or calculate a score for each player and keep a league table similar to that used in the 'multiplication competition' (Chapter 1).

3 Experiment with different techniques for creating the effect of a 'laser shot'. For example, drawing and deleting a dotted line would be quicker than printing and deleting a large number of characters. Now try the effect of obeying the two statements

```
VDU 19, 0,7, 0,0,0
VDU 19, 0,0, 0,0,0
```

The instantaneous change of actual background colour from black to white and back again creates a 'gunflash' effect.

4 Arrange for an 'explosion' when an invader is hit. The explosion could be represented by two or three user-defined characters displayed in quick succession.

5 Arrange for the invaders to send out an occasional 'missile' which drifts down the screen towards the player's gun, forcing him to take avoiding action.

6 Add sound effects to your program.

4.4 Controlling movement within a maze

Many arcade games involve movement that is restricted te part of the screen. The commonest example of this type of game is the extremely popular 'Pacman'. Here movement takes place within a maze. A program controlling such animation needs to keep a record of which regions of the screen am prohibited and this record must be kept in such a way that the program can recognise very quickly that a particular character position is out of bounds. We could very easily represent a maze on the screen (in MODE 1 or 4 say) marking the walls of the maze with one character and the paths with another. The program could then store a record of the shape of the maze in a two-dimensional array with one location for each character position on the area of the screen being used. (We shall omit the bottom line of the screen from our maze as this makes it easier to avoid accidental scrolling.) The array declaration is:

```
DIM maze%(39, 30)
```

Each location in this array could contain a value (TRUE or FALSE say) indicating whether or not the corresponding character position was part of a path or part of the maze walls. However, although simple, this representation is rather wasteful of memory space. It occupies 1240 BASIC variables, each of which occupies 4 memory locations - nearly 5K of memory altogether.

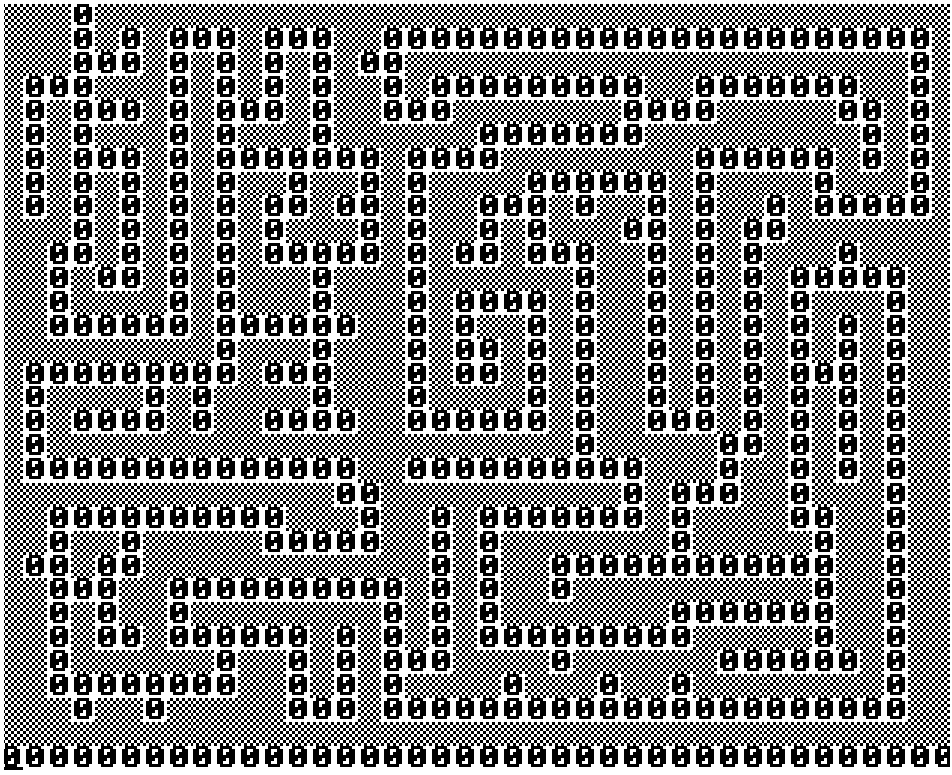
A much more appropriate representation for this sort of information is a bit map where the status of each character position on the screen is recorded as a single bit. Each BASIC variable occupies four 8-bit words, 32 bits in all, and so we can pack the information about one column of character positions into a single BASIC variable. We can represent the complete maze by a one-dimensional array:

```
DIM bitmap%(39)
```

where each location in this array will contain information about one column of character positions. This representation of the maze occupies only 160 bytes of memory - a considerable saving. Of course, such a saving would lose some of its value if it slowed down the process of recognizing the status of a character position on the screen. However, by careful use of logical operations (see below) we can avoid too much loss of speed.

Manual construction of maze

We can construct a maze by hand and code it up as a pattern of ones and zeroes where we use ones to represent the maze wall, and zeros to represent the paths. For example

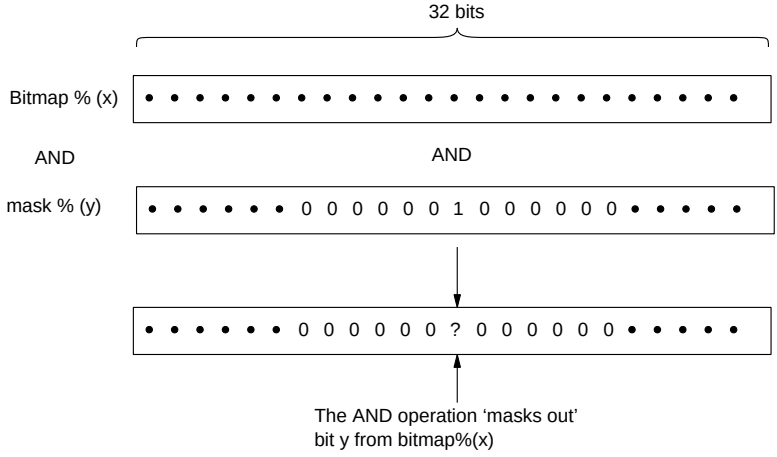


Remember that one column of the binary representation of the maze is to to be stored in a single word of our 'bitmap' array.
We have included a row of zeros corresponding to the bottom row of the screen which is not used. The easiest way of inputting the above maze to a program is to code each column reading from bottom to top in hex (Appendix 2). A pattern of 32 bits can be represented as a group of 8 hexadecimal digits. Thus 39 8-digit hexadecimal numbers, one for each column of the maze, can be supplied to a program in DATA statements. Here is a short program to draw a maze from such data.

```
10  MODE 1
20  VDU 23,224,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
30  VDU 23;8202;0;0;0;
40  DIM bitmap%(39),mask%(30)
50  PROCsetmasks
60  PROCdrawmaze
70  K=GET : END
```

```
   80  DEF PROCsetmasks
   90    mask%(0)=1
  100    FOR m=1 TO 30
  110      mask%(m)=mask%(m-1)*2
  120    NEXT m
  130  ENDPROC

  140  DEF PROCdrawmaze
  150    LOCAL x,y,path$,wall$
  160    path$=" " : wall$=CHR$(224)
  170    FOR x=0 TO 39:READ bitmap%(x) :NEXT
  180    COLOUR 1
  190    FOR y = 0 TO 30
  200      FOR x=0 TO 39
  210        IF bitmap%(x) AND mask%(y) THEN PRINT wall$;
             ELSE PRINT path$;
  220    NEXT:NEXT
  230  ENDPROC
 1000  DATA  &7FFFFFFF, &7F707E07, &601743F7, &4ED55800,
             &685557AB, &6B155029, &4FD45FFF, &68D74001,
             &6AD47FFD, &62D70001, &7AD7DFAF, &7A9558A1,
             &42B55A3D, &5EB40381, &42A5DABF, &7E8FF83B,
             &40FFFFE1, &57F4002D, &5015FFA5, &5FF50BB5
 1010  DATA  &58152895, &4BD5EED5, &5BD40855, &5257FB55,
             &5B500255, &4B57FF55, &5B47FD45, &5B7C006D,
             &490DFFED, &5D6C0025, &5563FFB5, &557801B5,
             &557FFCB5, &554007B5, &501F7635, &57F012E5,
             &5FFFF68D, &400006FD, &7FFFFE01, &7FFFFFFF
```

This program uses a logical AND operation at line 210 to test whether position x,y in the maze is marked with a 1 sepresenting the maze wall.



To do this we need to test whether bitmap%(x) has a 1 in bit position y. This is done by 'making' bitmap%(x) with a bit-pittern or 'mask' that contains only a single 1 in position y and zeros everywhere else. The result of the AND operation between bitmap%(x) and mask%(y) is non-zero only if there is a 1 in position y of bitmap%(x).

The array containing the masks needed in the above operation is initialised by PROCsetmasks which makes use of the fact that multiplying an integer by 2 corresponds to shifting the corresponding bit-pattern along one place.

A maze design program

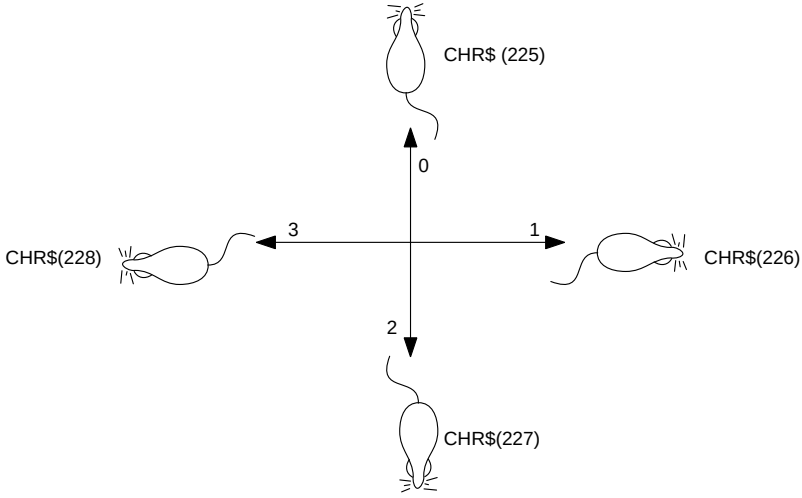
Before we look at the problem of moving objects around within the maze, here is a program that will make it easier to design a maze. It starts with a screen full of colour (the maze wall colour). You can move around the screen with keys L(eft), R(ight), U(p) and D(own). Switch to path drawing mode with P, switch to wall drawing mode with W and switch to nuzve mode (where you can move around without changing the maze) with M. To set a bit to 1 in the bit map, the maze design program uses an OR operation with the appropriate mask and to set a bit tot) in the bit map, an AND NOT operation is performed. These both appear at line 490. See Appendix 2 for further details on logical operators.

```
 10  MODE 1
 20  VDU 23,224,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
 30  VDU 19,2,11,0,0,0
 40  VDU 23;8202;0;0;0;
 50  DIM bitmap%(39) ,mask%(30)
 60  PROCsetmasks
 70  PROCconstructmaze
 80  MODE 7 :PRINT "10000 DATA &";~bitmap%(0);
 90  FOR i=1 TO 19:PRINT ",&";~bitmap%(i);:NEXT
100  PRINT' "10010 DATA &";~bitmap%(20);
110  FOR i=21 TO 39:PRINT ",&";~bitmap%(i);:NEXT
120  PRINT: END
130
140  DEF PROCsetmasks
150    mask%(0)=1
160    FOR m=1 TO 30
170      mask%(m)=mask%(m-1)*2
180    NEXT m
190  ENDPROC

210  DEF PROCconstructmaze
220    COLOUR 129
230    VDU 28,0,30,39,0 : CLS : VDU 28,0,31,39,0
240    COLOUR 128
250    FOR y = 0 TO 39 : bitmap%(y)=&FFFFFFFF : NEXT
260    x=0:y=0:wall=FALSE:moving=TRUE
```

```
  270    star$ = "*" + CHR$(8)
  280    wall$ = CHR$(224) + CHR$(8)
  290    path$ = " " + CHR$(8)
  300    COLOUR 2 :PRINT TAB(x,y);star$; :COLOUR 1
  310    REPEAT
  320      command$=GET$
  330      PROCprocess(command$)
  340    UNTIL command$="F"
  350  ENDPROC

  370  DEF PROCprocess(c$)
  380    IF INSTR("LRUDWPM",c$)=0 THEN ENDPROC
  390    IF bitmap%(x) ANDmask%(y) THEN PRINT wall$;
  400    IF c$="L" THEN IF x>0 THEN x=x-1
  410    IF c$="R" THEN IF x<39 THEN x=x+1
  420    IF c$="U" THEN IF y>0 THEN y=y-1
  430    IF c$="D" THEN IF y<30 THEN y=y+1
  440    IF c$="W" THEN wa11=TRUE :moving=FALSE
  450    IF c$="P" THEN wal1=FALSE:moving=FALSE
  460    IF c$="M" THEN moving=TRUE
  470    COLOUR 2:PRINT TAB(x,y); star$;:COLOUR 1
  480    IF moving THEN ENDPROC
  490    IF wall THEN
           bitmap%(x)=bitmap%(x) OR mask%(y)
         ELSE bitmap%(x)=bitmap%(x) AND NOT mask%(y)
  500  ENDPROC
```

A maze-running mouse

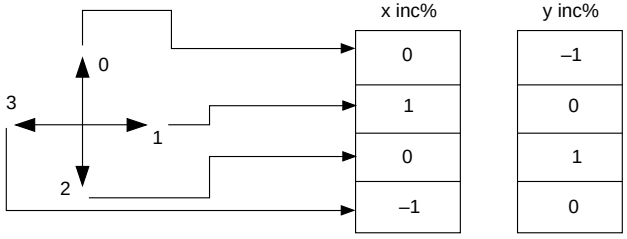
We now present a program that draws the same maze as before and controls a mouse as he runs about exploring the maze.



In applications like this, it is often necessary to design more than one user-defined character for the object being animated so as to be able to display the object in different orientations. In this case we have defined four versions of the mouse pointing in the four different directions in which he can move. These directions are numbered 0, 1, 2, and 3.

The mouse starts in the middle and its position and orientation are represented by its x-y coordinates ('mx%', 'my%') and a direction code (mdir%) which is set to 0, 1, 2 or 3.

The program makes the mouse explore the maze by repeatedly calling PROCmove and it is the definition of this procedure that determines the mouse's general behaviour. In the first version of the program (lines 470 to 510), at each step the mouse takes, this procedure counts in 'n%' the number of directions in which the mouse can move (excluding the direction from which it has just come). The possible directions are listed in the array 'possdir%'. The value of n% determines the action taken. If n% = 0, then it has reached a dead end and must turn back the way it came. If n% = 1 then it moves in that one direction, otherwise it makes a random choice from the directions available. Note the use of arrays 'xinc%' and 'yinc%' which are used to quickly convert direction codes (0, 1, 2 or 3) into x and y increments for a direction.



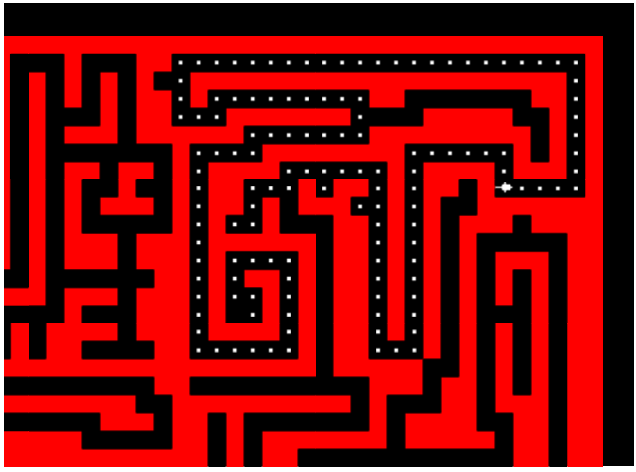
An attempt has been made to make the behaviour of the mouse more realistic by inserting time delays at appropriate points. For example it pauses when it has a choice of routes. The position and duration of such delays is worth experimenting with.

```
10  MODE 1
20  VDU 23,224,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
30  VDU 23,225,&10,&38,&38,&7C,&38,&10,&10,&10
40  VDU 23,226,0,8,&1E,&FF,&1E,8,0,0
50  VDU 23,227,&10,&10,&10,&38,&7C,&38,&38,&10
60  VDU 23,228,0,&10,&78,&FF,&78,&10,0,0
70  VDU 23;8202;0;0;0;
80  DIM bitmap%(39) ,mask%(30)
90  DIM xinc%(3), yinc%(3), possdir%(3)
```

```
100  xinc%(0)=0 : xinc%(1)=1 : xinc%(2)=0 : xinc%(3)=-1
110  yinc%(0)=-1 : yinc%(1)=0 : yinc%(2)=1 : yinc%(3)=0
120  PROCsetmasks
130  PROCdrawmaze
140  PROCputmouseinmiddle
150  PROCexplore
160  K=GET
170  END

180  DEF PROCsetmasks
190    mask%(0)=1
200    FOR m=1 TO 30
210      mask%(m)=mask%(m-1)*2
220    NEXT m
230  ENDPROC

240  DEF PROCdrawmaze
250    LOCAL x,y,path$,wall$
260    path$=" " : wall$=CHR$(224)
270    FOR x=0 TO 39 :READ bitmap%(x):NEXT
280    COLOUR 1
290    FOR y = 0 TO 30
300      FOR x=0 TO 39
310        IF bitmap%(x) AND mask%(y) THEN PRINT wall$;
           ELSE PRINT path$;
320    NEXT:NEXT
330    COLOUR 3
340  ENDPROC

350  DEF PROCputmouseinmiddle
360    mx%=20 : my%=15 : mdir%=0
370    mouse$=CHR$(225)
380    PRINT TAB(mx%,my%); mouse$
390  ENDPROC

400  DEF PROCexplore
410    REPEAT
420      PROCmove
430    UNTIL my%=0
440  ENDPROC

450  DEF PROCmove
160  LOCAL fromdir%,newdir%,n%,d%
170    fromdir% = (mdir%+2) MOD 4
180    n% = 0
190    FOR d%=0 TO 3
500      IF d%<>fromdir% THEN
           IF (bitmap%(mx%+xinc%(d%))
           AND mask%(my%+yinc%(d%))) = 0
           THEN n%=n%+1 : possdir%(n%)=d%
510  NEXT d%
```

```
  520     IF n%=0 THEN newdir%=fromdir%
          ELSE IF n%=1 THEN newdir%=possdir%(1)
          ELSE PROCdelay(20) : newdir%=possdir%(RND(n%))
  530     PROCturn(newdir%)
  540     PROCstep
  550   ENDPROC

  560   DEF PROCturn(d%)
  570     IF d%=mdir% THEN ENDPROC
  580     mdir%=d% : mouse$=CHR$(225+d%)
  590     PRINT TAB(mx%,my%);mouse$;
  600     PROCdelay(10)
  610   ENDPROC

  620   DEF PROCstep
  630   LOCAL nx%, ny%
  640     nx%=mx%+xinc%(mdir%)
  650     ny%=my%+yinc%(mdir%)
  660     PRINT TAB(mx% ,my%); " "; TAB(nx%,ny%);mouse$;
  670     mx%=nx% : my%=ny%
  680   ENDPROC

  690   DEF PROCdelay(d)
  700   LOCAL t
  710   t=TIME+d
  720   REPEAT : UNTIL TIME>t
  730   ENDPROC

 1000   DATA ... as before
```

The next photograph shows part of the maze in which the area explored by the mouse has been marked with 'droppings'.



A straightforward modification to the program was used to obtain this display.

```
 74  VDU 23,229,0,0,0,&18,&l8,0,0,0
 76  dropping$ = CHR$(229)
       :
       :
660  PRINT TAB(mx%,my%); dropping$;
     TAB(nx%,ny%); mouse$;
```

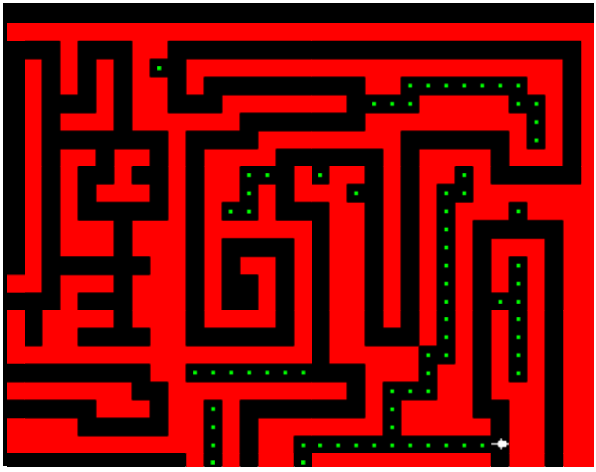
We have simply replaced the space used to delete the mouse when it is being moved by a character consisting of a white spot.

Marking dead ends

The only use made of the bit map so far is to mark the position of the walls of the maze. However, we could easily wake the program extend the prohibited region, while the program was running, by adding ones to the bit map on paths that have been found to be dead ends. We recognise a dead end at line 520 (n% = 0) and can set a variable 'deadend%' to TRUE. When there is a choice of paths (n% > 1) we can set deadend%' to FALSE. We then modify PR0Cstep so that if deadend%' is TRUE, then the square being nxsved from is marked in the bit map as prohibited. Because 'deadend%' remains TRUE until a choice of paths is encountered, all squares down the dead end will be marked on the way out. We ran also modify the program so that the 'droppings' are left only in the dead ends.

```
 75  VDU 19,2,2,0,0,0
       :
       :
520    IF n%=0 THEN deadend% = TRUE:newdir%=fromdir%
       ELSEIF n%=1 THEN newdir%=possdir%(1)
       ELSE deadend%=FALSE : PROCdelay(20):
         newdir%=possdir%(RND(n%))
       :
       :
630  LOCAL nx%,ny%,dropcol%
635  IF deadend% THEN
       bitmap%(mx%) = bitmap%(mx%) OR mask%(my%) :
       dropcol%=2
     ELSE dropcol%=0
       :
       :
660  COLOUR dropcol% : PRINT TAB(mx%,my%);dropping$;
661  COLOUR 3 : PRINT TAB(nx%,ny%);mouse$;
```

The spot is now printed in COLOUR 2 to mark dead ends, or in COLOUR 0 elsewhere. Printing a character in COLOUR 0 (the background colour) has the same of feet as printing a space.



Exercises

1 Experiment with time delays in the mouse program to make him appear hesitant in different places.

2 Make the mouse look down each direction that is recognised as a possibility when he is considering which way to go.

3 Add sound effects to the mouse program, for example a 'frustrated squeak' when it hits a dead end and an 'excited squeak' if it reaches the exit. (Perhaps the exit could be made more interesting by adding a a user-defined character to represent a piece of cheese.)

4 The mouse frequently returns to his 'den' in the centre after exploring part of the maze. Make him curl up and sleep in the corner of the den whenever this happens. (You will need to define one or two characters to represent a sleeping mouse.)

5 Change the mouse program so that the mouse is controlled from the keyboard with keys telling it to turn left, right, up or down. Record the time taken by a user at the keyboard to find the way out of the maze. (Note that we have defined PROCdelay in such a way that it does not alter the variable TIME.)

4.5 Animating line drawings

Up to now we have looked at animation using characters and this is the most commonly used mechanism in microcomputer animation. It has spawned a vast industry of computer games, and such character animation techniques must be the moat commonly viewed computer image. In many applications, however, the use of character animation is inconvenient and we may wish to compose 'frames' of an animated sequence by drawing lines. This may be because we wish to animate a sequence that is mathematically defined (the cross section of a piston engine driving a crankshaft, for example), or because we want to animate using frames that have been drawn by hand on a graphics tablet. In either case the source material will be a list of coordinates and the most convenient tool to deal with a list of coordinates is the PLOT statement.

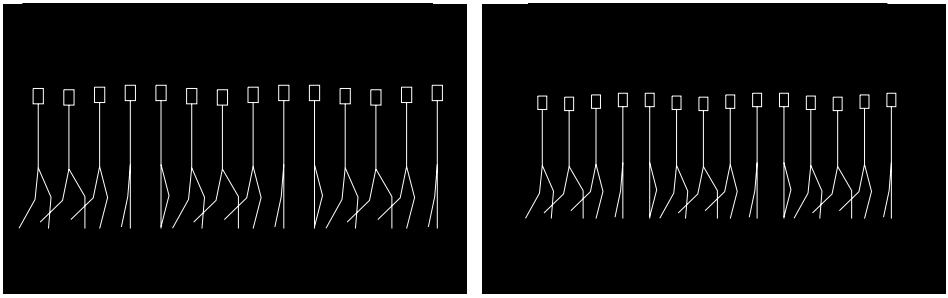
When film cartoons are made by hand, animated effects are created by drawing and photographing a large number of frames which are then displayed by a projector at a speed that gives the impression of continuous movement.

Computer animation packages now exist that help the cartoon artist to create animated films. Such a package typically includes programs that help the artist to design scenes from the film, using commands for interactively drawing lines and colouring regions. An animation package also includes programs for carrying out tasks such as 'in-betweening', a tedious and time-consuming job carried out by the 'in-betweeners' or junior artists of the cartoon film industry. Here, the main frames of a film are created on the screen by an artist and the hundreds of in-between frames that bridge the gaps between the main frames are generated by the animation package, each frame being photographed as it is created.

If you have the facilities for making films and wish to use your BBC micro for creating cartoons, then the full power of the graphics facilities can be used in drawing each frame of the film. The time taken to change the image on the screen is not critical as each frame of the film will be photographed only when the changes on the screen are complete. It can take many hours of program runs to create a few seconds of film in this way.

In this section we will look at techniques for animating tine drawings. Although the graphics facilities on the BBC micro are extremely powerful and versatile, they are generally too slow for the animation of large objects drawn with line drawing and colour fill facilities. However, line drawings such as simple stick figures can be fairly successfully animated in real time.

Frames for animating a stick man
In order to animate a line drawing we need to draw a sequence of separate frames representing the object in different stages of movement. There are many ways in which these frames could be presented to the computer, but, in our Illustrative example of line drawing animation, we shall present each frame in the form of a DATA statement vontaining a list of coordinates that describe a stick man. We shall animate the man so that he appears to walk across the screen. The frames that we shall use are displayed simultaneously in the first photograph.



There are in fact only five different frames, which are displayed repeatedly. In the interests of brevity, we have omitted the arms. We will create the required walking effect by displaying and then deleting successive representations of the man, each one being displayed a little further across the screen than the previous one. Note that the vertical height of the head varies depending on the way in which the legs are bent. This effect is exaggerated in the second photograph and such exaggeration could be used to put more of a 'spring' in his step.

Representing frames for stick figures
The DATA statement for a frame will contain 9 values. The first is a y-increment to give the rise and fall of the body from frame to frame. The back is always in the same orientation and so it need not be specified for each frame. The next 8 values in a frame DATA statement represent four x-y pairs. These are

(1) the coordinates of the first knee relative to the top of the leg,

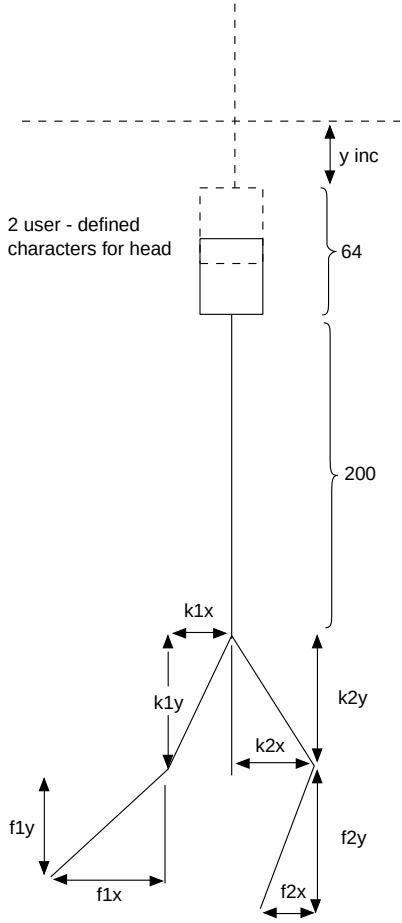
(2) the coordinates of the first foot relative to the first knee,

(3) the coordinates of the second knee relative to the top

of the leg, and

(4) the coordinates of the second foot relative to the second knee.

We use the following nomenclature:

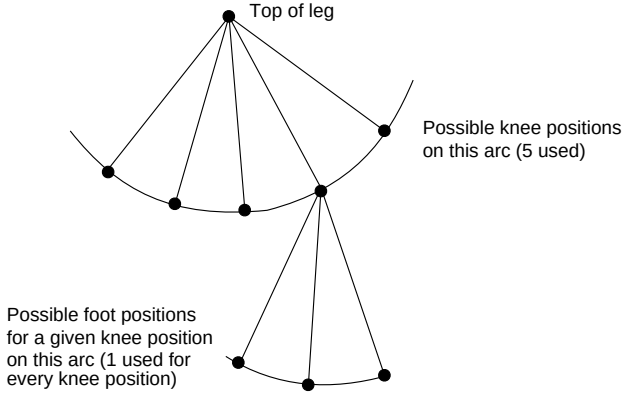


In the program, values representing the five frames will be stored in parallel arrays, where the frames are numbered 0 to 4. The 'yinc' values for a frame is not stored but is used to calculate a y-coordinate for starting to draw the frame and a y-coordinate for the top of the legs. It is these two values that are stored along with the other coordinates £ran the DATA statement for a frame. The head is

drawn by printing two user-defined characters. We use the VDU 5 statement to arrange that characters are printed at the current graphics position.

Designing frames for stick figures

When calculating coordinates for stick figures, it is necessary to bear in mind that distances between joints shuld not vary from frame to frame. The possible knee coordinates relative to the top of the leg were constructed by marking the top of the leg on a piece of graph paper and using that point as the centre of a circle whose radius was the distance from the top of the leg to the knee.



Associated with each knee position is a set of possible foot positions. These can also be obtained by drawing a circle of appropriate radius centred on the required knee position.

Animation by repeated deleting and drawing

Here is a first attempt at a program that makes the stick man walk across the screen. It works by repeatedly deleting and redrawing the man. Deletion is achieved by 'drawing' in the background colour. The flickering effect which is the main drawback of this program will be rectified shortly.

```
10  DIM k1x%(4), k1y%(4), f1x%(4), f1y%(4),
        k2x%(4), k2y%(4), f2x%(4), f2y%(4),
        hy%(4), ly%(4)
20  y% =600
30  FOR f=0 TO 4
40  READ  yinc%, k1x%(f), k1y%(f), f1x%(f), f1y%(f),
          k2x%(f), k2y%(f), f2x%(f), f2y%(f)
50  hy%(f)=y%+yinc% : ly%(f)=hy%(f)-264
60  NEXT f
```

```
   70  VDU 23,224,0,0,0,0,&FF,&81,&81,&81
   80  VDU 23,225,&81,&81,&81,&81,&81,&81,&81,&FF
   90  head$=CHR$(224)+CHR$(8)+CHR$(10)+CHR$(225)
  100  MODE 1 : VDU 5
  110  x%=16 : xinc%=16
  120  frame%=0
  130  PROCdrawman(0,x%,0,3)

  140  REPEAT
  150    nx%=x%+xinc%
  160    PROCdrawman(frame%,x%,0,0)
  170    frame%=(frame%+1) MOD 5
  180    PROCdrawman(frame%,nx%,0,3)
  190    x%=nx%
  200  UNTIL x%>1200
  210  VDU 4
  220  MODE 7
  230  END

  300  DEF PROCdrawman(f%,x%,l%,c%)
  310  LOCAL ly%
  320    GCOL l%,c%
  330    MOVE x%-16,hy%(f%)
  340    PRINT head$; : PLOT 0,-16,-32
  350    DRAW x%,ly%(f%)
  360    PLOT 1,k1x%(f%),k1y%(f%) :
         PLOT 1,f1x%(f%),f1y%(f%)
  370    MOVE x%,ly%(f%)
  380    PLOT 1,k2x%(f%),k2y%(f%) :
         PLOT 1,f2x%(f%),f2y%(f%)
  390  ENDPROC

 1000  DATA -10,40,-92, -8,-98,-10,-99,-50,-90
 1010  DATA -14,50,-86,0,-100,-20,-98,-70,-68
 1020  DATA -6,25,-97,-25,-97,-20,-98,-70,-68
 1030  DATA 0,0,-100,0,-100,-8,-100,-20,-95
 1040  DATA 0,25,-97,-25,-97,0,-100,0,-100
```

The man is drawn (or deleted) by PROCdrawman which accesses one of the sets of values stored in the 'frame arrays'. The first parameter of the procedure selects the frame to be drawn. The second parameter specifies the x-coordinate (the y-coordinate is stored as part of the frame). The remaining two parameters specify the logical plotting operation and colour code to be used in drawing the man. In this program, the colour is either 3 (for draw) or 0 (for delete). The logical plotting operation is always 0, but this parameter is needed in the next version of the program.

Image plane switching

The flickering effect exhibited by tho above program was due In the fact that we could see the man being erased and rvdrawn. In order to eliminate this flickering effect, we noed to arrange for the erasing and redrawing process to take place invisibly. To do this, we need to work with two neparate 'image planes' and display one plane on the screen while the erasing and redrawing process is being carried out In the other plane.

In MODE 1, the colour of each pixel is coded as a two-bit number. We saw in Chapter 2 that, instead of treating the screen as a single image plane in which each pixel is one of four colours, we can treat it as two separate image planes In which each pixel is one of two colours. In each MODE 1 pixel, one of the two bits is taken to represent the colour of a pixel in one plane and the other bit is taken to represent the colour of the corresponding pixel in the other plane. The alternative significance of each two-bit colour code is given by the following table:

Single image plane		Two separate image planes	
colour code	bit pattern	plane 1	plane 2
0	00	0	0
1	01	1	0
2	10	0	1
3	11	1	1

To switch between planes 1 and 2, we use VDU 19 statements to associate different combinations of actual colours with our four colour codes. For example, if we want 0 to be the background colour code and 1 to be the foreground colour code in each of planes 1 and 2, then we can selectively display one of the two planes by selecting one of the two actual colour combinations given in the following table:

Colour code	Actual colour settings	
	plane 1 displayed plane 2 hidden	plane 2 displayed plane 1 hidden
0	background	background
1	foreground	background
2	background	foreground
3	foreground	foreground

If we are using the same background and foreground colours in plane 1 as in plane 2, colour code 0 is always set to the background colour and colour code 3 is always set to the foreground. If the background colour is black and the foreground colour is white, then the colour codes 0 and 3 are correctly initialised in MODE 1. To switch plane 1 on

and plane 2 off, we need only use:

```
VDU 19, 1,7, 0,0,0
VDU 19, 2,0, 0,0,0
```

and to switch plane 1 off and plane 2 on, we use

```
VDU 19, 1,0, 0,0,0
VDU 19, 2,7, 0,0,0
```

A new shape can be plotted in plane 1 by preceding the plotting instructions by

```
GCOL 1,1
```

A shape can be erased from plane 1 by replotting it after

```
GCOL 2,2
```

(see Chapter 2). Similarly, a shape can plotted in plane 2 by preceding the plotting instructions by

```
GCOL 1,2
```

and erased by redrawing the shape after

```
GCOL 2,1
```

The following is an outline of how we use the above technique to conceal the deleting and redrawing process while an object is being moved about the screen:

```
Set x,y to the initial position of the object
Switch plane 1 on, plane 2 off
Draw first fraurs in plane 1

REPEAT
  Calculate newx,newy
  Draw next frame at newx,newy in off plane
  Switch planes
  Erase frame at position x,y in plane that is now off
  x=newx : y=newy
UNTIL final position reached
```

The next program fills in the details needed to make our man walk across the screen. Note that the speed at which the man walks can be varied by changing the x-increment (xinc%) between frames. (For very fast or slow motion, it may be necessary to change the stride length in the 5 basic frames used.) An alternative way of slowing him down is, of course, to insert a delay loop.

```
        .
        .
        .
110   x$=16 : xinc%=16
120   frame%=0
130   PROCswitchon(1)
140   PROCdrawman(0,x%,1,on%)
150   REPEAT
160     nx%=x%+xinc% : nf% = (frame%+1) MOD 5
170     PROCdrawman(nf%,nx%,1,off%)
          : REM hidden draw in new position
180     PROCswitchon(off%)
190     PROCdrawman(frame%,x%,2,on%)
          : REM hidden delete in off frame!
200     x% = nx% : frame% = nf%
210   UNTIL x%>1200
220   VDU 4
230   k=GET : MODE 7
240   END

300   DEF PROCdrawman(f%,x%,l%,c%)
        :
      as before
        :
390   ENDPROC

400   DEF PR0Cswitchon(screen%)
410   on% = screen% : off% = 3-on%
420   VDU 19, on%, 7, 0,0,0
430   VDU 19, off%,0, 0,0,0
440   ENDPROC
        :
        :
```

Exercises

1 Extend The DATA statements used to represent the frames for our stick man so that arm positions can be specified. Extend PROCdrawman accordingly.

2 Add feet to the stick man.

3 Multiply the y-increment values by a 'bounce factor' and experinment with the effects obtained.

4 Give the man a dog on a lead.

5 Assign a set of frames for a stick-horse and make it walk, trot or gallop across the screen.

4.6 Palette changing

The 'palette' of actual colours associated with the colour codes for a mode can be changed instantaneously with the VDU 19 statement and we have already used this statement in several programs in this chapter as well as in Chapter 2. In 'The BBC Micro Book' we demonstrated how palette changing could be used to animate, spinning disks, for example. We finish the present chapter with a further demonstration of the use of this technique to create the illusion of movement. We shall use the same stick man as we used in the last section, but this time we shall create an army of stick men marching across the screen. To produce this effect, we need to have available a colour for each different frame well as a background colour. In this case, we need at least six colours and we must therefore run the program in MODE 2. (This causes a slight change in the shape of the man because af the different resolution.)

The program starts by setting colours 0 to 5 to black, selecting colour 5 as the background colour (GCOL 0, 133) and clearing the screen. The program then cycles frames 0 to 1 as before, drawing them at success positions on the screen, but this time frame 0 is drawn in colour 0, frame 1 in colour 1 and so on. At this stage the men are invisible. The animation effect is now created by cycling through the colours 0 to 4, at each step using VDU 19 statements to switch the previous colour to black and the next colour to white. This creates the impression that a succession of men is continually marching across the screen.

```
        :
        :
 70   VDU 23,224,0,0,0,0,&F,9,9,9
 80   VDU 23,225,9,9,9,9,9,9,9,&F
 90   head$=CHR$(224)+CHR$(8)+CHR$(10)+CHR$(225)
100   MODE 2 : VDU 5
110   x%=16 : xinc%=48
120   FOR c=0 TO 5:VDU 19,c,0,0,0,0:NEXT
130   GCOL 0,133 : CLG

140   frame%=0
150   REPEAT
160     PROCdrawman(frame%,x%,0,frame%)
170     frame%=(frame%+1) MOD 5
180     x%=x%+xinc%
190   UNTIL x%>1220
```

```
200   frame%=0
210   REPEAT
220     nf%=(frame%+1) MOD 5
230     VDU 19,frame%,0,0,0,0
240     VDU 19,nf%,7,0,0,0
250     PROCdelay(10)
260     frame%=nf%
270   UNTIL INKEY$(0)=" "
280   VDU 4 : K=GET : MODE 7 : END

300   DEF PROCdrawman(f%,x%,l%,c%)
        :
      as before
        :
390   ENDPROC

400   DEF PROCdelay(d)
410     LOCAL t
420       t=TIME+d
430       REPEAT : UNTIL TIME>t
440   ENDPROC

        :
      DATA as before
        :
```

Exercises

1 Create the effect of a bouncing ball by drawing a number of balls in different vertical positions (non-overlapping) and in different colours, and then using palette changing.

2 Create the effect of a rotating sphere by first drawing a circle and then drawing lines of longitude in different colours. Then use palette changing to reveal each line of longitude in turn.