# ZX Spectrum Game Master

PK McBride




Longman

# ZX Spectrum
# Game Master

# ZX Spectrum Game Master

PK McBride

Longman

ZX SPECTRUM is a Trade Mark of
SINCLAIR RESEARCH LIMITED.

Longman Group Limited
Longman House, Burnt Mill, Harlow,
Essex CM20 2JE, England
and Associated Companies throughout the world.

© Longman Group Limited 1984

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the Copyright owner.

First published 1984

ISBN 0 582 91606 2

Printed in UK by Parkway Illustrated Press, Abingdon

Designed, illustrated and edited by Contract Books, London

The programs listed in this book have been carefully tested, but the publishers cannot be held responsible for problems that might occur in running them.

# Contents

**Introduction** 6

**Action games** 12
1 Movement 16
2 Targets & invaders 36
3 The Hall of Fame 48
4 Mazes 52
5 Special effects 62
Program listings 68

**Adventure games** 76
6 The spirit of adventure 76
7 BASIC logic 80
8 Planning the game 84
9 The key routines 98
10 Taking it further 106
Program listing 114

**Interactive games** 122
11 Game design 122
12 First moves 124
13 Developing tactics 128
14 Take it from here 132
Program listing 136

**Appendices**
A Essential BASIC 148
B Defining your own graphics 154
C Super screens 157

# INTRODUCTION



Why bother to write your own computer games? After all, it is hard work, and there are so many games around already that you may wonder whether it is really worth the effort. Of course it is, because however much hard work is involved, writing a game is an exciting challenge. When you have succeeded in producing the game you want, that does the things you want it to do, in the way that you choose, there is an enormous sense of satisfaction. It's a bit like climbing a mountain. It may be a struggle getting there, but there's a great view from the top.

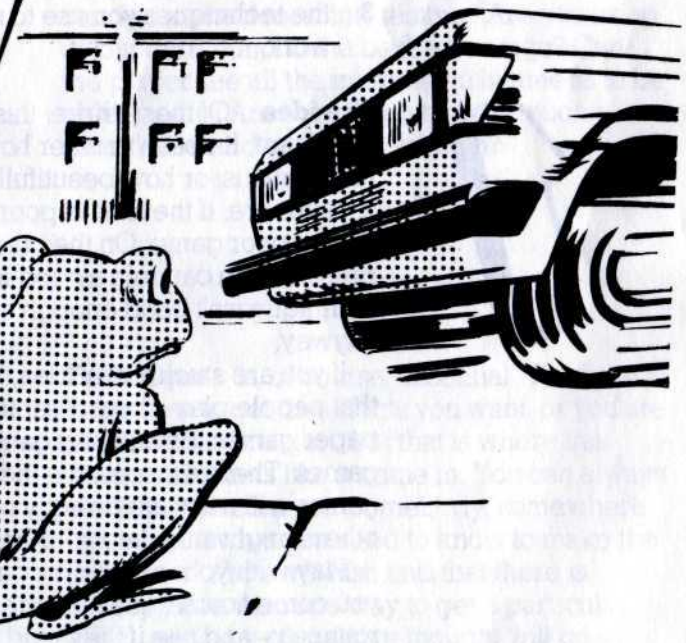
And are there so many games around already? How many truly original games have you come across? How many are simply new versions of old favourites? Could you come up with something really new? If you could, then there's someone out there who would like to play it, and perhaps, to pay for the pleasure.

The second question is 'Why BASIC?' Machine code programming produces much, much faster games – about fifty times as fast as BASIC programming. The answer there is that BASIC is far easier to use, and do you really need the speed? Those all-action high-speed arcade games represent weeks or months of work by teams of experienced programmers.



You want to see results, don't you? Also, there are many types of computer games where high-speed is simply not necessary. You don't need it in adventure games or games of strategy. Imagination is far more important here, and imagination and attractive graphics will go a long way to make up for lack of speed in action games.

This is the real strength of BASIC programming. You will find that you spend less time struggling with the technicalities of the program, and more time using your imagination. You can always add machine code routines into your BASIC programs – short, tried-and-tested routines that will speed up particular parts of the program or give you special effects that are not possible in BASIC. Useful routines can often be found in computing magazines.



# THE GAME

There are three aspects to any computer game:

1. the idea
2. the way you present the game
3. the techniques you use to make the game work.

## 1

**The idea** Of these three, this is the most important. It doesn't matter how good your technique is, or how beautifully designed your screens are, if the idea is poor, then you still have a poor game. On the other hand, poor presentation can ruin a good idea, and if your technique isn't good enough, it won't work anyway.

If you are stuck for an idea, look at the games that people play – board games, pencil and paper games, plastic peg games, group activity games. There's inspiration there. Some of these games will translate directly onto a computer, others might stimulate your imagination.

Play with your micro. Try out anything new you come across, any new ideas, techniques or routines – and see if they lead to a game of some sort.



## 2

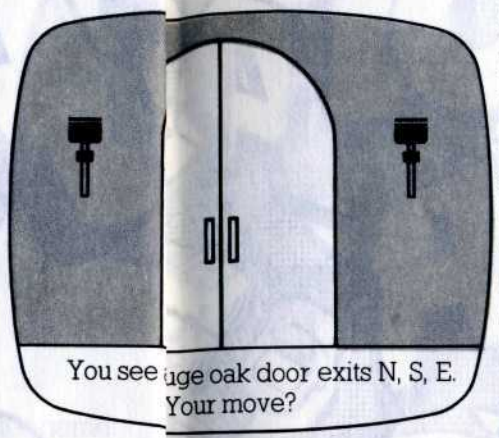
**Presentation** comes next. How will your game actually appear on the screen? What will the player have to do to play the game? How can you make it most interesting? Are you using colour and sound to the best advantage? Can the player see all the information he needs to be able to play? Are you giving him too much information and causing confusion?

## 3

**Technique** is, of course, essential. You have to know how to get the effects you want, or you are stuck before you start, but that is where this book, and others like it come in. You can always read up on technique. Somebody, somewhere, can tell you what you need to know to make the game work. You will also find that there is usually more than one way to get a particular effect, and a bit of creative thought will go a long way.

Computer games come in a bewildering variety of types, shapes and sizes. Some are true computer games – the sort that don't exist in any other form – others are 'computerised games' – versions of existing board or paper games.

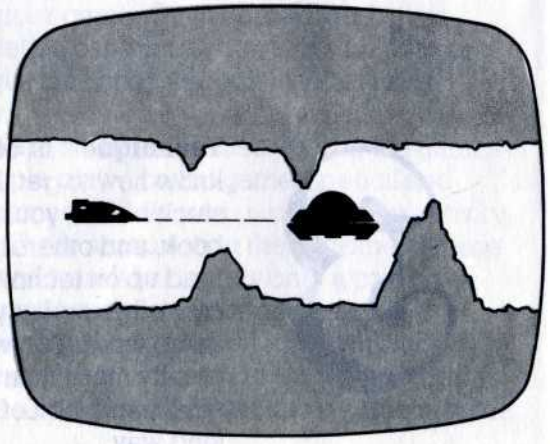
The book is divided into three sections – Action, Adventure, and Interaction. It is a convenient way to try to cover the techniques of games-writing, but in practice many games refuse to fit into nice neat categories.




## Action games

These are the games played out on the screen – the arcade games. They have come a long way in a few short years, from the TV Tennis of the early 70s through Space Invaders, Munchman, Donkey Kong, Frogger to the latest 3-D effect Star Wars games. All action games are based on speed of movement and of response, but speed is not the only thing.

The best games have excellent graphics and sound effects, and it's here that imagination comes into play.

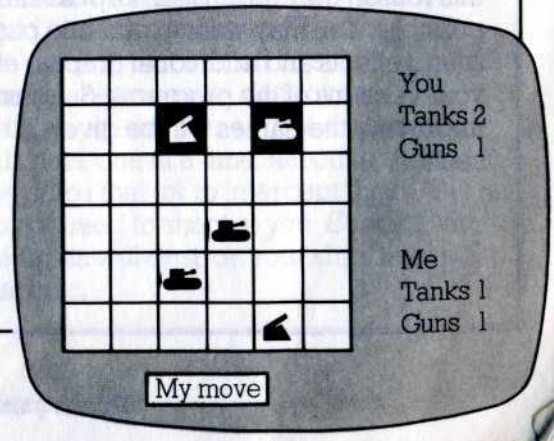


## Adventure games

The classic adventure games – developed on mainframe computers in the 60s and 70s – were all text based. Games were often included in commercial software packages, just to show off how well the computers could handle words. Cheap chips have brought to home computers the same kind of memory capacity that the old mainframes had, and now adventure games have reached the home. Many of the newer games include graphics, sometimes in the form of illustrations, sometimes as action games spliced into the main adventure. All adventure games share a common theme – the player has to find his way around some unknown land, using his wits to solve the many problems he meets, trying to discover the hidden treasures.

## Interactive games

All computer games are interactive one way or another, i.e. the computer responds to what you do, and you must respond to the computer. All games must work this way. By 'interactive games' I mean those where the computer acts as a player, in much the same way as a human would act. The machine has to 'think', to make decisions and to work through a strategy. These types of games require very careful planning, and even the simplest interactive game is far from simple to write. This section comes at the end of the book for that reason!



All action games depend upon speed of response. The speed with which the player responds to the computer, and the speed with which the computer responds to the player. This can create problems when you are working in BASIC.

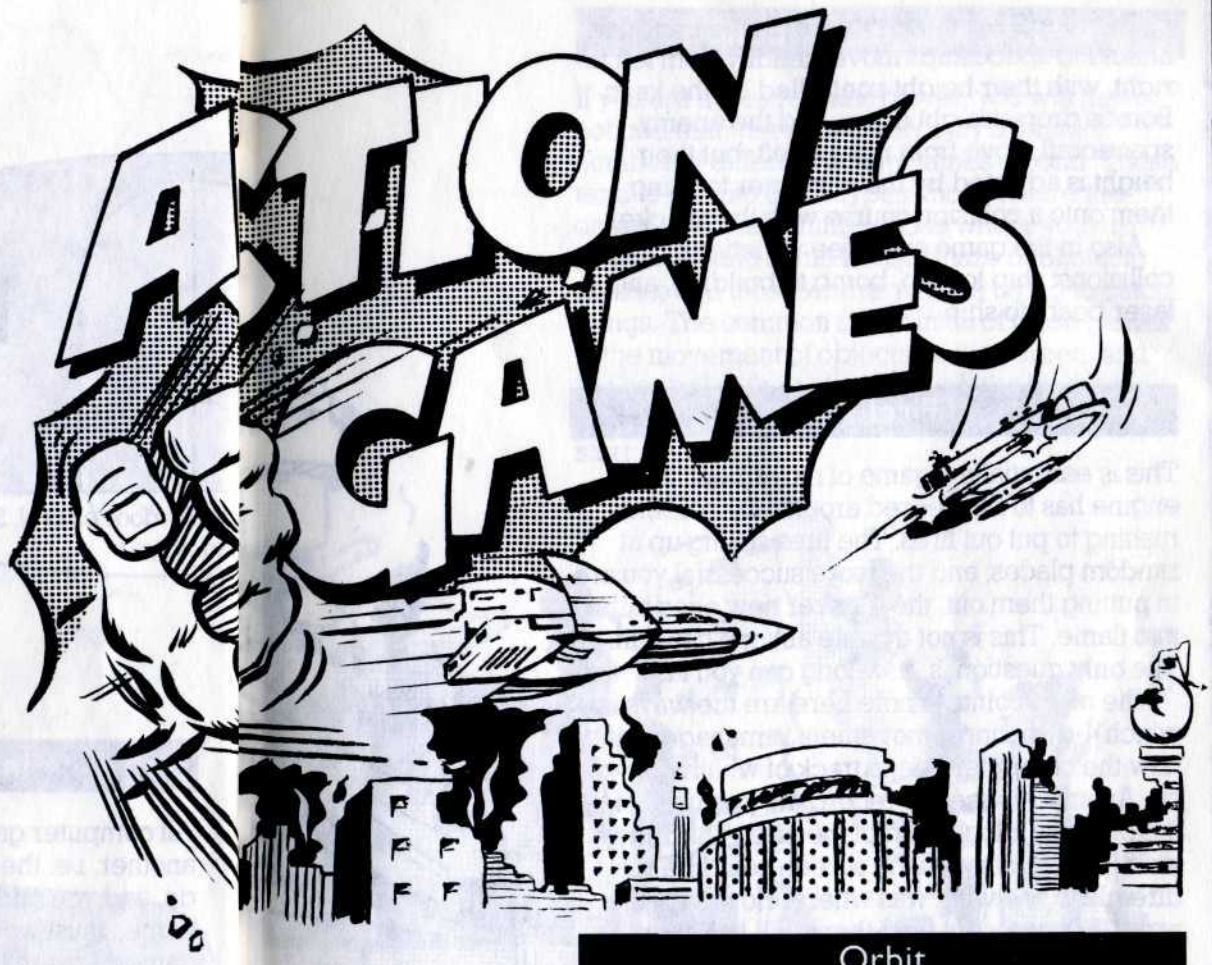
Don't be put off by this. You can achieve some quite slick games in BASIC, as long as you keep the routines simple. Also, BASIC does have one enormous advantage over machine code. It is fairly simple to handle, and not too hard to track down your mistakes. Debugging machine code is no fun at all!

## The listed games

You will find the lists for two examples of action games at the end of this section of the book. These could be typed in now, or you may prefer to wait until you have worked your way through the section. If you wait, then you will have a better understanding of how the games work, and will be less likely to make mistakes as you type them in. On the other hand, typing them in now will give you something to refer to as you read.

The purpose of the games programs is to demonstrate techniques and routines, and for this reason they have been kept as simple as possible. You may want to add title pages, sound effects and additional graphic effects to your versions of the programs. Suggestions for improving the games will be given in this section.






## Orbit

This game looks at simple movement, and shooting and bombing routines. The story behind the game is that a small fleet of spaceships has been sent to destroy the buildings on an enemy planet. The ships make their bomb runs, one at a time, shooting down any enemy ships that try to intercept them. The enemy do not need to shoot at you. Contact with their force fields will destroy your ship, leaving them unharmed.

The game has three types of movement. The attacking spaceships move steadily from left to right, with their height controlled by the keys. Bombs drop straight down, and the enemy spacecraft move from right to left, but their height is adjusted by the computer to bring them onto a collision course with the attacker.

Also in the game are three varieties of collisions: ship to ship, bomb to building, and laser beam to ship.

## Firefight

This is essentially a game of movement. A fire engine has to be steered around the screen, rushing to put out fires. The fires spring up at random places, and the more successful you are in putting them out, the quicker new ones burst into flame. This is not a game anyone can win. The only question is, how long can you last?

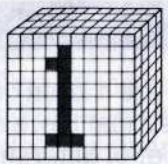
The main points to note here are the ways in which 4-directional movement is managed, and how the computer keeps track of what's going on. An array is used to record the positions of the obstacle blocks and of the fires. This is not strictly necessary, as the screen could be read directly to see what was where. However, the array is convenient, and there will be other times when you will need to use one to store this type of information.



## Shoot, steer and catch

If you are a keen games player, you will have noticed that there are really only a limited number of different action games around. These tend to fall into three types: those where the object is to shoot things; those where you are steering round some kind of maze or obstacle course; and those where you are trying to catch things. The common factor in all of these games is the movement of objects on the screen, and the control of the action through the keyboard (or joysticks.) This then is where we should start.





# Movement

## Key Moves

This shows the essence of key control. You need two variables to store the position of your object (line 10). You must display your object on screen (line 20). The computer must scan the keyboard to find out if any keys have been touched (line 30). Lastly you need to translate the key contacts into changes in the position variables (lines 40 to 70). And, of course, you have to keep doing it (line 80).

```
10  LET L=10 : LET C=15
20  PRINT AT L,C;"*"
              (use any character you fancy)
30  LET A$=INKEY$ : IF A$ = "" THEN GOTO 30
40  IF A$="7" THEN LET L=L-1
50  IF A$="6" THEN LET L=L+1
60  IF A$="8" THEN LET C=C+1
70  IF A$="5" THEN LET C=C-1
80  GOTO 20
```

## Which keys?

I have used the 'arrow' keys for the controls in this program. You may not like them. Pick the keys you would like to use instead, and type their characters into lines 40 to 70 to customize your program. You can, if you like, include a routine in your programs which lets each player choose his own keys for controlling movement. To do this you need to create a small array to hold the player's choices, collect those choices, and then compare A$ with the array. The 'D.I.Y. Key Routine' does this.

This program will crash if you go off the screen. Simply re-RUN to begin again.

## D.I.Y. key routine

```
 5  GOSUB 1000
10  ....
20  ....          (as opposite)
30  ....
40  IF A$= K$(1) THEN.... up
50  IF A$= K$(2) THEN.... down
60  IF A$= K$(3) THEN... right
70  IF A$= K$(4) THEN.... left
80  GOTO 20
1000 DIM K$(4)   (the Key array)
1010 INPUT " WHICH KEY FOR
     UP ?"; K$(1)
1020 INPUT " WHICH KEY FOR
     DOWN ?"; K$(2)
1030 INPUT " WHICH KEY FOR
     RIGHT ?"; K$(3)
1040 INPUT " WHICH KEY FOR
     LEFT ?"; K$(4)
1050 RETURN
```

Notice how the key choice routine has been made into a subroutine, even though it is only used once in each game. This helps to speed up the computer's responses. Every time it hits a GOTO line, it has to work out where the line is by going back to the start of the program. The fewer lines there are at the top of the program, the quicker it calculates its GOTO.

## Continuous movement

The Key Moves program doesn't give you continuous movement – it produces a continuous line instead. To give the impression that an object is moving, the first essential is that there should be only one image on the screen at any time. To do this, you must rub out the old image before the new image is printed. The change of variables doesn't have to be through key controls. A target, for instance, would probably move smoothly across the screen. Here's a target moving routine.

```
1000  REM TARGET
1010  FOR C=0 TO 31
1020  PRINT AT 5,C;"*"
1030  PRINT AT 5,C;" "
1040  NEXT C
```

Type it in and run it.

Now there's a target few people would be able to hit! BASIC can be very fast, as long as the routine is short and simple. You will need to slow the target down, to give people a chance to see it. Insert this line:

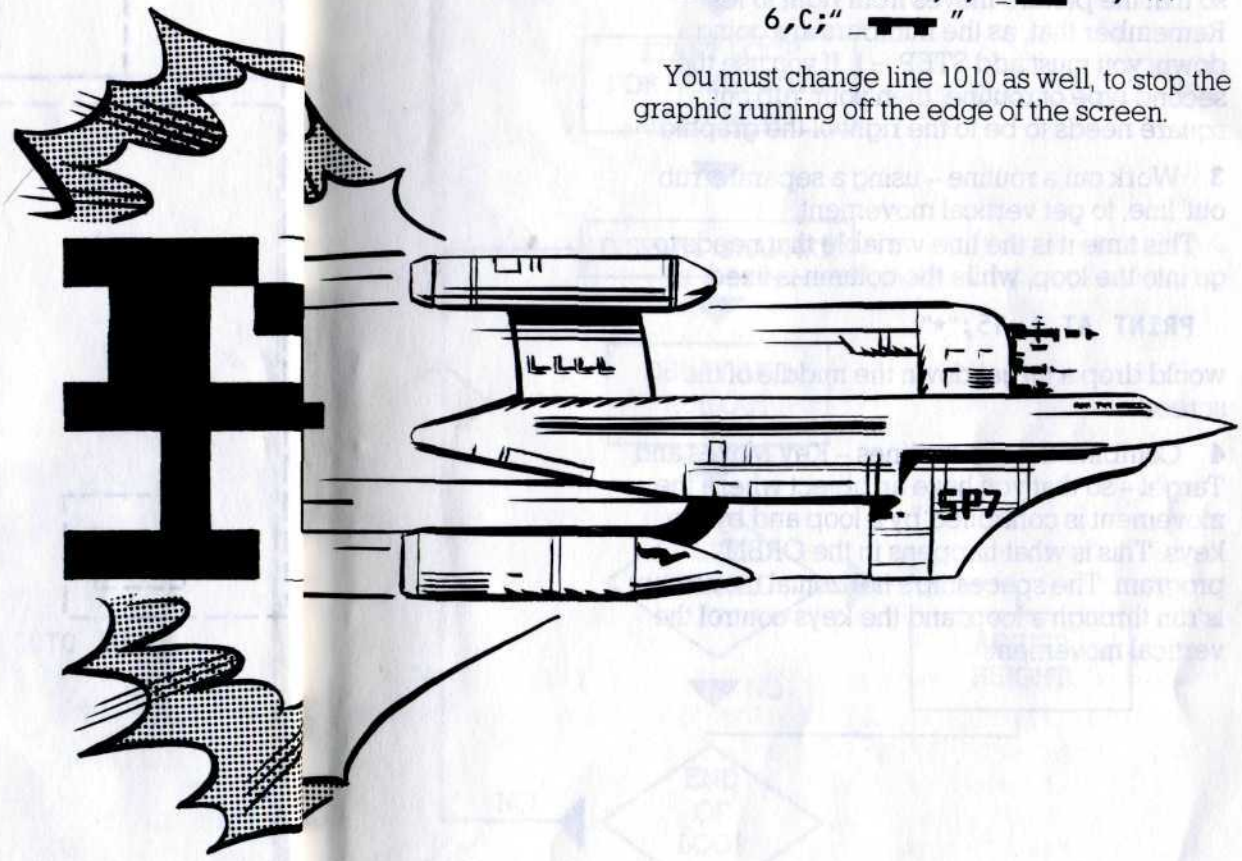
```
1025  FOR D=1 TO 50 : NEXT D
```

This is a delay loop. The computer isn't actually doing anything at all, but it is doing it 50 times, and that slows things down a bit. Change the number in the loop to change the speed.

Here's another way to move a target. It's a simpler routine, but it works perfectly well as long as the movement is straight across the screen.

```
1010  FOR C = 0 TO 30
1020  PRINT AT 5,C;" *"
1030  FOR D=1 TO 50 : NEXT D
1040 NEXT C
```





Rewrite line 1020 in that last example to give a bigger picture – 6 squares, rather than one. You will notice, when you run this, that the movement has slowed down a little and has become a bit jerky.

```
1020  PRINT AT 5,C;"  ■■  "; AT
      6,C;"  ■■■■  "
```

You must change line 1010 as well, to stop the graphic running off the edge of the screen.

## Things to try

**1** Add a GOTO line to keep the movement going endlessly.

**2** Alter the numbers in the FOR. .NEXT. . loop so that the picture moves from right to left. Remember that, as the numbers are going down, you must add STEP −1. If you use the second type of routine, then your 'rub out' square needs to be to the right of the graphic.

**3** Work out a routine – using a separate 'rub out' line, to get vertical movement.

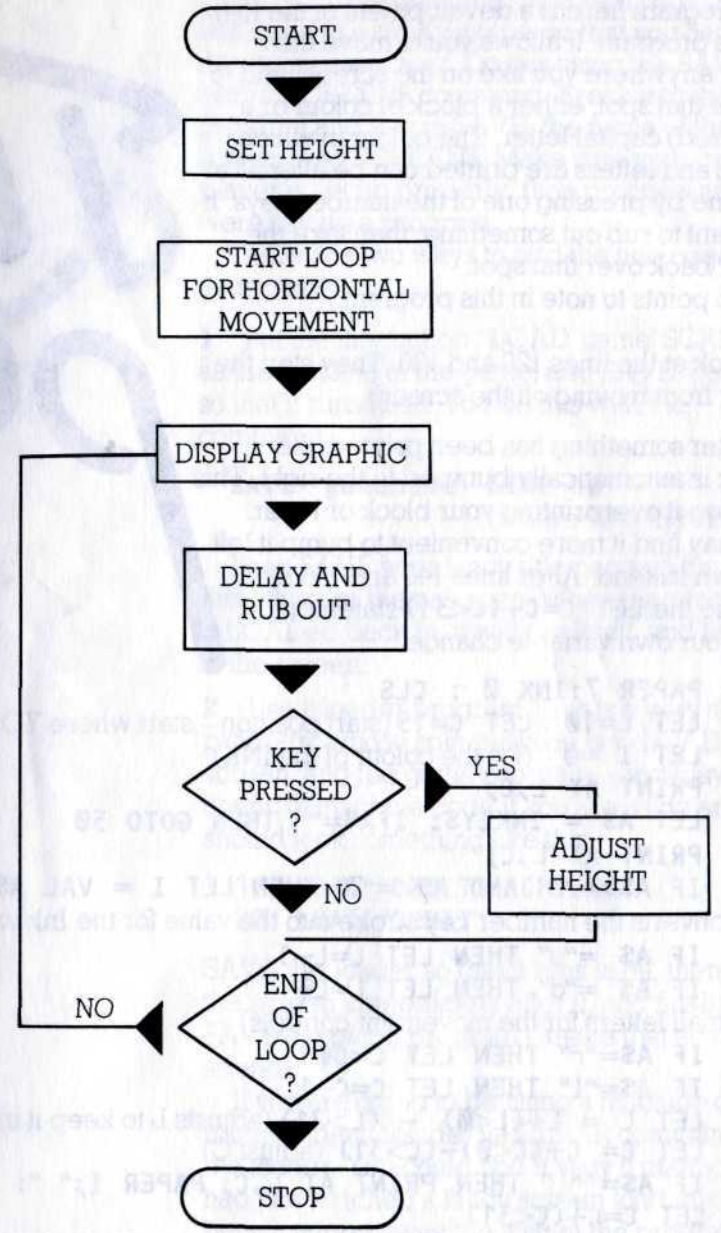
This time it is the line variable that needs to go into the loop, while the column is fixed.

```
PRINT AT L,15;"*"
```

would drop a target down the middle of the screen.

**4** Combine the two routines – Key Moves and Target – so that you have an object where the movement is controlled by a loop and by the keys. This is what happens in the ORBIT program. The spaceship's horizontal movement is run through a loop, and the keys control the vertical movement.





# Title pages

The program here is a development of the Key Moves program. It allows you to move the cursor anywhere you like on the screen, and to print at that spot, either a block of colour or a (coloured) capital letter. The colour in which blocks and letters are printed can be altered at any time by pressing one of the number keys. If you want to rub out something, then take the cursor back over that spot.

Two points to note in this program.

**1** Look at the lines 120 and 130. They stop the cursor from moving off the screen.

**2** After something has been printed, the cursor is automatically bumped to the right. This is to stop it overprinting your block or letter. You may find it more convenient to bump it left or down instead. Alter lines 140 and 150, and replace the `LET C=C+(C<31)` statement with your own variable change.

```
10  PAPER 7:INK 0 : CLS
20  LET L=10 : LET C=15 (start position – start where YOU want)
30  LET I =0  (I is the colour of the INK)
40  PRINT AT L,C; "*"
50  LET A$ = INKEY$: IFA$="" THEN GOTO 50
60  PRINT AT L,C;" "
70  IF A$>="0" AND A$<="7" THEN LET I = VAL A$
 (converts the number key stroke into the value for the Ink variable)
80  IF A$ ="u" THEN LET L=L-1
90  IF A$ ="d" THEN LET L=L+1
 (small letters for the movement controls)
100  IF A$="r" THEN LET C=C+1
110  IF A$="l" THEN LET C=C-1
120  LET L = L+(L<0) - (L>21) (adjusts L to keep it in range)
130  LET C= C+(C<0)-(C>31) (adjust C)
140  IF A$= " " THEN PRINT AT L,C; PAPER I;" ":
     LET C=C+(C<31)
150  IF A$>="A" AND A$<="Z" THEN PRINT AT L,C;INK I;
     A$: LET C=C+(C<31)
160  GOTO 40
```



Once you have designed your title page, you will need to save it on tape, so that you can add it to your game later. Do this using the SAVE 'name' SCREEN$ command. Break into the program and key in SAVE, the name of the screen, and SCREEN$. Make sure your cassette player is set up properly, then proceed as if you were saving a program.

There are two ways to add the title page to your game.

**1** Put the instruction : LOAD 'name' SCREEN$ as the first line of the game, and SAVE the game so that it runs itself. You do this with the command

```
SAVE "gamename" LINE 10
```
(where 10 is the first line)

Then SAVE your fancy title page on the tape directly after the program. When the program is LOADed back in, it will run itself, and LOAD in the screen.

**2** Use a loader program. This is a very short program, whose sole purpose is to LOAD in a screen, and the game program – and maybe a chunk of machine code if you are using any. It should look something like this:

```
10  LOAD "ORBIT" SCREEN$
20  LOAD "ORBIT"
```

SAVE this loader so that it runs itself, then SAVE the screen immediately after it, and finally SAVE the game program – make that auto-run as well.

If you are going to be using a machine code routine, then include the CLEAR command in the loader. You could find it very irritating if you had just watched a fancy screen load in, only to have it wiped away the instant the program started.

```
5  CLEAR 56999
```
(or whatever numbers)

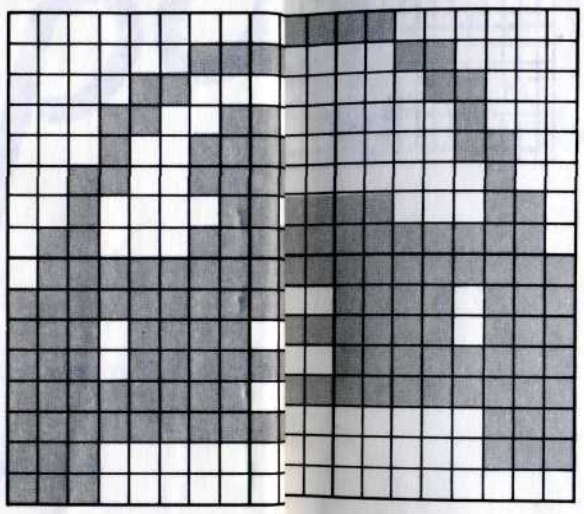
# Brakes out!

Despite its name, this is not a version of the wall game, but one where you have to steer a runaway car down a hill – and your brakes are not working.

The car only moves from side to side (controlled by keys [5] and [8]). The downhill movement is created by making the screen scroll upwards by continually printing on the bottom line.

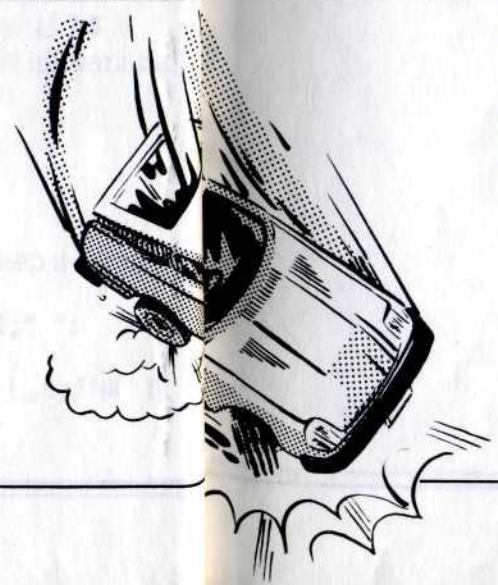
Normally, if you try to print below line 21, you will get the 'scroll?' message. In this program, the SCRLCT (Scroll Count) systems variable has been reset, so that it will print an additional 255 lines before the 'scroll?' appears (line 140).

You can improve the car design by changing the data in the subroutine at 1000. The original design is shown for your reference.



## Main loop

```
100  GOSUB 1000
110  CLS : LET T=10
120  LET SC=0 : LET C =14
125  REM GRAPHICS IN 130
130  PRINT AT 10,C;"ABC";AT
     11,C;"DEF"
140  POKE 23692,255
200  GOSUB 2000
210  LET A$=INKEY$
220  LET C1=C+(A$="8")-(A$="5")
230  IF ATTR (11,C1) =0 OR ATTR
     (11,C1+2)=0 THEN GOTO 400
240  LET C=C1
245  REM GRAPHICS IN 250
250  PRINT PAPER 8;AT 9,C-1;
     " (5 spaces) "; AT 10,C-1;
     " ABC ";AT 11,C;"DEF"
260  FOR D=1 TO 25 : NEXT D
270  LET SC=SC+1
280  GO TO 140
```



```
400  PRINT "YOU DROVE ";SC/10;"
     KM" ,"BEFORE YOU CRASHED."
410  STOP
```

## Graphics

```
1000  FOR N=0 TO 5: READ G$
1010  FOR R=0 TO 7: READ B
1020  POKE USR G$ +R,B
1030  NEXT R: NEXT N
1040  DATA "A",0,1,6,12,9,25,16,49
1050  DATA "B",0,255,0,192,224,224,
      192,255
1060  DATA "C",0,128,96,48,16,24,
      8,140
1070  DATA "D",63,127,119,119,127,
      127,112,112
1080  DATA "E",255,255,129,255,129,
      255,0,0
1090  DATA "F",252,254,238,238,254,
      254,14,14
1100  RETURN
```

Lines 1040 to 1090 contain the DATA for each of the graphics A, B, C, D, E and F.

## Kerb printing

```
2000  LET X=INT(RND * 3)-1
2010  LET T1=T+X
2020  IF T1>3 AND T1<19 THEN LET
      T=T1
2030  PRINT AT 21,T;PAPER 0;"■";
      AT 21,T+10;"■":PRINT
2040  RETURN
```

Most of what's happening here should be fairly obvious, but the following points are worth looking at more closely.

## The winding road

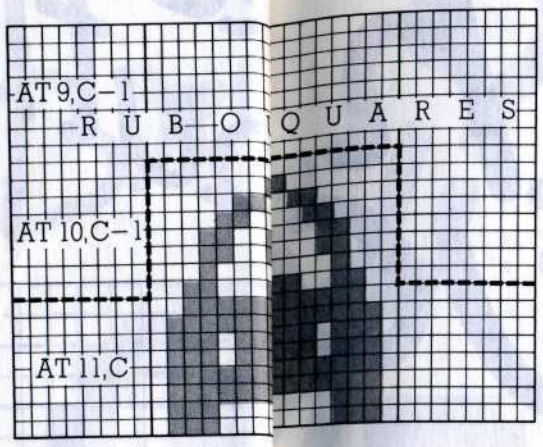
The road edge must change gradually, no more than one column either way at a time – in other words, the T variable must be altered by −1,0 or +1. Line 2000 produces a suitable random number. INT(RND*3) gives 0,1 or 2. Take away 1 to drop it into range.

## Testing for crashes

The kerbstones don't just look black, they are also printed on black PAPER. This means that the colour ATTRIBUTES of the kerb squares are quite different from the rest of the screen. The ATTR(Line, Column) function will tell you what colours are used at any particular square on the screen, using this code: ATTR number = INK code, plus 8 times the PAPER code, plus 64 if the square is BRIGHT, and 128 if it is FLASHing. The normal state of the screen is white PAPER and black INK, so the ATTRibutes of a square would be 8*7+0=56. Where the PAPER is black, the ATTRibutes are 0*8+0=0. Line 230 tests the squares where the wheels would be printed on the next move, to see if either are kerb squares.

## Rub out

There is no separate rub out line here. The car is printed (line 250) with extra spaces above and round the top line. These will rub out the old image, even though the new image may not be directly in line.






## Try this

**1** If you haven't already done so, now is the time to design your own car for this game. It doesn't have to be a car. It could be a downhill skier, a parachutist landing in a narrow ravine, a spaceship lowering down into the depths of a forbidden planet, or anything else your imagination will stretch to.

**2** Give the screen a different appearance. Instead of marking the edges of the road, you could mark the road itself:

```
2030 PRINT AT 21,T;
     PAPER 0; " (12 spaces) "
```

Now check it to see if the car has lost contact with the road:

```
IF ATTR(11,C1)<>0 OR
ATTR(11,C1+2)<>0....
```

Don't forget to draw the road at the start, and to print the car directly onto it in a light colour.

**3** Why have a road at all? Why not random obstacles coming up at the car/skier/boat/spaceship? This is actually easier. All you need is a routine to produce a random number between 0 and 31, and to print something on the chosen column.

You could even arrange it so that your score depended upon how many of those obstacles/objects you landed on.

Play around with the scrolling screen idea and see what you can come up with.

# Which way?

In those games where the playing piece can move in different directions about the screen, it looks much better if you have suitable graphics for the directions. You can see this in a simple form in the FIREFIGHT game. There, the fire engine faces left, right, front or back as appropriate.

It all makes extra work, of course. For a start, you need more user defined graphics. A typical set for 8-directional movement is shown here. As graphics go, they are not very beautiful, but at least there is no doubt as to which way they point. Define them as a set, and in the right order, so that the graphic for direction 1 is in Graphics 'A', direction 2 in Graphics 'B' and so on. This is important, as it makes changing direction very simple.

The directions are coded 1 to 8. To make the correct graphic appear, you need a line like this:

```
PRINT AT L,C;CHR$(143 +D)
```

(CHR$ 144 is Graphics 'A', 145 is Graphics 'B'.)



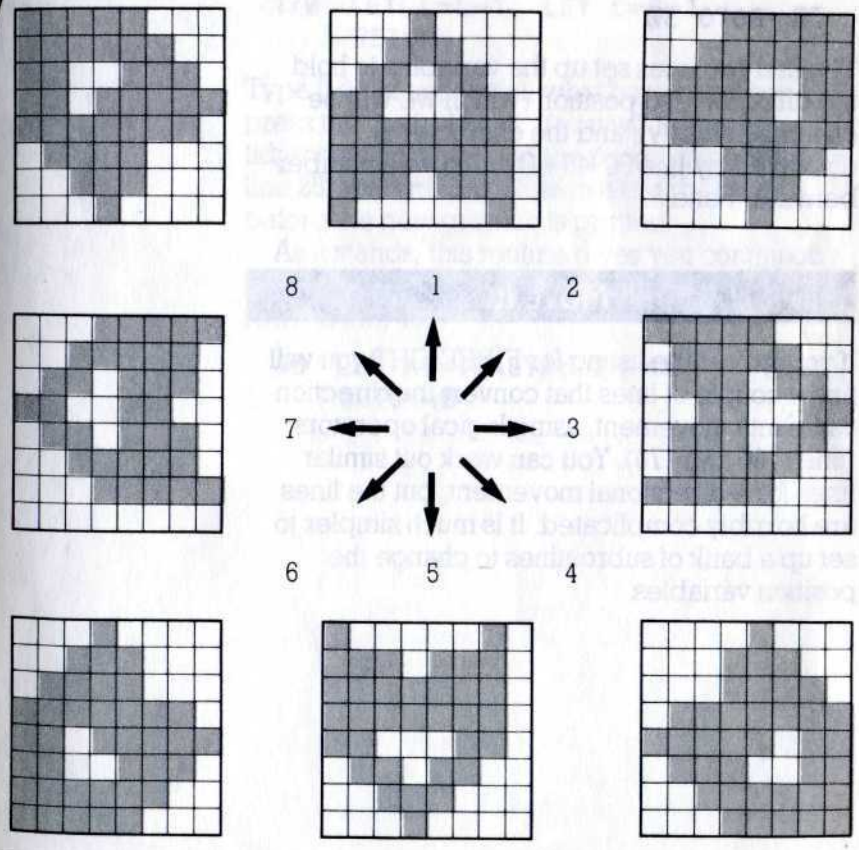
## Key controls

You could have a different key for each direction – the number keys 1 to 8 would seem appropriate – but that is not terribly user friendly. It would be simpler, from the player's point of view, to have one key for a clockwise turn, and one for an anticlockwise turn.

```
1000  IF A$="5" THEN LET D=D-1
1010  IF A$="8" THEN LET D=D+1
```

These can be combined into one line using logical operators.

```
1000  LET D=D+(A$="8")-(A$="5")
```



A further line is needed to stop D wandering out of range. This line allows for full travel either way round; you can go clockwise from 8 to 1, and anticlockwise from 1 to 8.

```
LET D=D+(8 AND D<1)-(8 AND D>8)
```

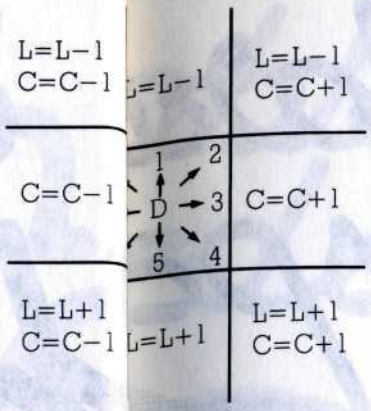
You can see the way this works from this program. Type it in and try it.

```
10  LET D=1
20  LET L=10: LET C=15
30  PRINT AT L,C;CHR$(48+D)
40  LET A$=INKEY$
50  LET D=D+(A$="8")-(A$="5")
60  LET D=D+(8 AND D<1)-(8 AND
    D>8)
70  GOTO 30
```

The first two lines set up the variables to hold the direction and position (which we will be changing shortly), and the character is displayed by line 30. 48+D will give a number between 1 and 8.

## Move it

If you look at the listing for FIREFIGHT you will find a couple of lines that convert the direction value into movement, using logical operators (lines 160 and 170). You can work out similar lines for 8-directional movement, but the lines are horribly complicated. It is much simpler to set up a bank of subroutines to change the position variables.




```
70  GOSUB (90 +10*D)
80  LET L=L +(22 AND L<0)-
    (22 AND L>21)
90  LET C =C +(32 AND C<0)-
    (32 AND C>31)
95  GOTO 30
100 LET L=L-1 : RETURN
110 LET L=L-1: LET C=C+1:RETURN
120 LET C=C+1 : RETURN
130 LET L=L+1: LET C =C+1 :
    RETURN
140 LET L=L +1: RETURN
150 LET L=L+1 : LET C=C-1:
    RETURN
160 LET C=C-1 : RETURN
170 LET L=L-1: LET C=C-1 :
    RETURN
```

Type it all in and watch what happens when you press the control keys. If you would prefer a tidier screen, then loop the program back to line 25, and write in a line to clear the screen before the new graphic is printed.

As it stands, this routine gives you continuous movement. To make it one step at a time, then rewrite line 40:

```
40  LET A$=INKEY$: IF A$ ="" THEN
    GOTO 40
```

When you have got only one object on the screen, it's simple enough to control its speed through a variable in a delay loop.

```
10  LET WHEN = 50
    ...
    ...
300  FOR D=1 TO WHEN : NEXT D
```

WHEN can then be increased or decreased by a key control line.

With two or more objects moving on screen, this method will not do. If you increase the delay to slow down one object, you slow down the whole program. The CATCH program given here shows one way to allow two objects to move at different, controlled, speeds. It works by using multiples of numbers, in the same way that the Fizz-Buzz game is played in schools.

## Fizz buzz

In Fizz-Buzz, pupils count round the class, one at a time, but when the number is a multiple of 3, the pupil says 'Fizz', rather than '3', '6', '9', or whatever. For multiples of 4, the pupils would say 'Buzz'. It's a fun way of learning tables. The Fizz-Buzz numbers don't have to be 3 and 4. Here's how it would sound when played with Fizz for 2, and Buzz for 5.

1, Fizz, 3, Fizz, Buzz, Fizz, 7, Fizz, 9, Fizz-Buzz, . . . .

The computer doesn't Fizz and Buzz. Instead, it goes to a subroutine to move the graphic, when it meets the right multiple in a simple counting loop. The overall loop (CT – Check Time) increases by 1 each time round. (Line 110). Each graphic has its own time code held in the T() array. Line 70 checks to see if the Check Time is a multiple of a graphic's time code. If it



# Varable speds



is, then the program goes off to move that graphic. The square block has a time code of 2, so that it is moved every second time the program goes round the loop. The angled block starts with a time code of 10, so that it is only moved every 10 loops. You can reduce this time code and speed up the graphic, by pressing [8]. [5] slows it down. The graphic can be moved up and down the screen by keys [6] and [7], and the object of the game is to match the speeds and positions of the two graphics.

Type the game in and get it working, then design some graphics of your own to make this look like a spacecraft docking manoeuvre. (For advice on using user-defined graphics, see Appendix B.)

```
10  REM CATCH
20  DIM T(2): DIM X(2): DIM Y(2):
    DIM C(2): DIM G(2)
30  LET T(1) =2: LET Y(1) =10:
    LET X(1) =0: LET C(1) =1:
    LET G(1) =143
40  LET T(2) =10: LET Y(2) =20:
    LET X(2) =0: LET C(2) =3:
    LET G(2) =142
50  POKE 23672,0: POKE 23673,0:
    LET CT=1: CLS
60  FOR N=1 TO 2
70  IF CT/T(N)=INT (CT/T(N)) THEN
    GO SUB 200
80  NEXT N
90  IF ABS (X(1)-X(2)) <2 AND
    Y(1) =Y(2) AND T(1) =T(2)
    THEN GO TO 120
100 LET A$=INKEY$: IF A$<>"" THEN
    GO SUB 300
110 LET CT=CT+1: GO TO 60
120 PRINT "SUCCESS IN ";INT ((PEEK
    23673*256+PEEK 23672)/50); "
    SECONDS."
```

```
130  STOP
200  PRINT AT Y(N),X(N);" "
210  LET X(N)=X(N)+1-(32 AND X(N)=31)
220  PRINT AT Y(N),X(N); INK C(N);
     CHR$ G(N)
230  RETURN
300  LET
     T(2)=T(2)+(A$="5")-(A$="8"):
     IF T(2) <1 THEN LET T(2) =1
310  LET YN=Y(2)+(A$="6")-(A$="7")
320  PRINT AT Y(2) ,X(2); " "
330  LET Y(2) =YN
340  LET N=2: GO SUB 220
350  RETURN
```

## Try this

**1** Improve the appearance of the game by giving it a background. If you are creating graphics to make this a spacecraft docking program, then add a subroutine to turn the screen black, and scatter stars at random over it. A few of these will get wiped out by the space station and satellite as they cross the screen, but it should not be enough to worry about.

**2** The movement here is horizontal only. This is to keep the program simple. Why not combine this program with the routines for multi-directional movement to make a more challenging and interesting game?

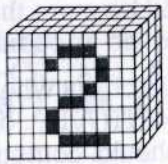
**3** The two objects do not have to start at any given place or speed. The initial values for T(1) and Y(1) could easily be randomized so that the game becomes a little different each time.

**4** Single square graphics are rather small, why not make them larger and more attractive? There will be a slight loss of speed, but this should be more than made up for by the improved appearance.

**5** Watch out when using lines like line 70. IF A/B =INT(A/B). . does not always work as well as it should. Sometimes you will find that it misses the odd multiple because of rounding errors. The computer works in binary numbers, and only converts these to decimal for your benefit. This rounding can occasionally lead to numbers being out by a very tiny decimal fraction. 9 and 9.000000001 may be, in reality, the same, but the computer will see them as two different numbers. A better check line in this kind of situation should take this form:

```
IF ABS( A/B - INT(A/B))<.0001
THEN.....
```

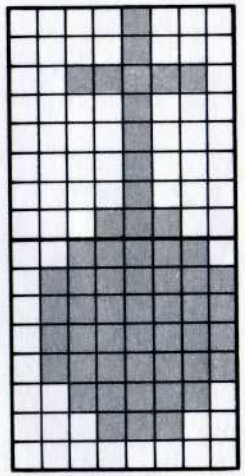
It now checks to see if they are 'equal enough', rather than exactly equal. You will see a similar check in line 90, where the question is, are the two objects close enough?



# Targets & invaders

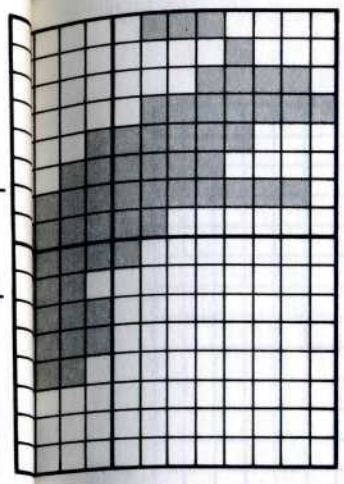
This shows a simple but effective way to run a shooting game. It uses high-speed bullets – so fast, indeed, that you can't see them. If you can't see them, then you don't have to program them, and that saves a lot of work.

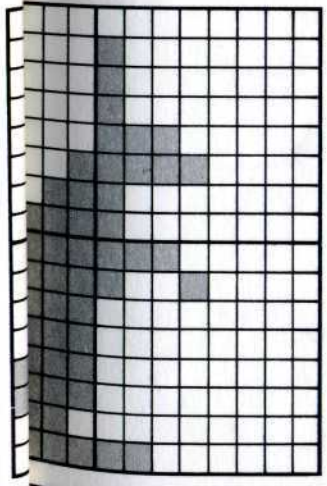
The player has a gunsight which he can steer around the screen using the keys around S. This key (S) acts as the trigger. If it is pressed when the gun is on target, then a hit is scored. To make life interesting, the targets (rabbits) appear on the screen at random heights and then hop (randomly) across.



```
 10  GOSUB 5000
 20  PAPER 4: BORDER 2: CLS
 24  REM GRAPHICS IN 25
 25  PAPER 8: PRINT AT GY,GX;
     "I";AT GY+1,GX;"J"
 30  LET RC=0: LET HIT=0
 40  LET RL=INT (RND*10)+5
 50  FOR D=1 TO 50: NEXT D
 60  LET J=INT (RND*3)
 70  IF J=0 THEN GO SUB 200: GO
     TO 50
 80  FOR I=RC TO RC+J
 85  REM GRAPHICS IN 90
 90  PRINT AT RL,I;" AB";AT
     RL+1,I; " CD"
100  GO SUB 200: IF HIT=1 THEN
     GO TO 140
110  NEXT I
```







```
120  LET RC=RC+J: IF RC>28 THEN
     GO TO 20
125  REM GRAPHICS IN 130
130  PRINT AT RL,RC;" EF";AT
     RL+1,RC; " GH": GO TO 50
140  LET T=INT (RND*500)
150  FOR D=1 TO T: NEXT D
160  GO TO 20
200  LET A$=INKEY$: IF A$="" THEN
     RETURN
210  PRINT AT GY,GX;" ";AT GY+1,
     GX; " "
220  LET GX=GX+(A$="D")-(A$="A")
230  LET GY=GY+(A$="X")-(A$="W")
235  REM GRAPHICS IN 240
240  PRINT AT GY,GX;"I"; AT GY+1,
     GX; "J"
250  IF A$="S" AND RC-GX<3 AND
     RL-GY<2 THEN GO TO 300
260  RETURN
300  PRINT AT 20,10; "GOT HIM!!"
310  FOR D=1 TO 500: NEXT D
320  PRINT AT 20,10;"  (9 spaces)   "
330  LET HIT=1: RETURN
5000 FOR N=1 TO 10: READ G$
5010 FOR R=0 TO 7: READ B: POKE
     USR G$+R,B: NEXT R: NEXT N
5020 DATA "A",0,0,0,0,1,3,23,63,"B",
     112,8,14,127,248,240,252,192,
     "C",63,14,15,7,30,48,96,64,
     "D",128,0,0,0,0,0,0,0
5030 DATA "E",0,0,0,0,0,1,3,7,"F",
     0,128,128,128,224,240,128,128,
     "G",15,15,31,95,255,255,126,55,
     "H",224,144,0,0,0,0,0,192
5040 DATA "I",8,8,62,8,8,8,8,28,
     "J",62,127,127,127,127,62,28,0
5100 LET GX=15: LET GY=20
5110 RETURN
```

## How it works

The main loop of this program runs from line 50 to line 130. This is the part that runs the rabbit across the screen. GOSUB 200, in lines 70 and 100, sends the program to the key controls subroutine. The size of the rabbit's jump is fixed by line 60, and could be of 1 or 2 squares or nothing. If 'nothing' then the hunter gets a chance to move his gun before the program goes back for the next jump.

Line 250 is the line that checks for hits. There are two parts to this – is the trigger being pressed, and is the gun on target? This gun obviously fires buckshot, as that line accepts close as being good enough to count. You could convert the gun to a rifle by insisting on greater accuracy.

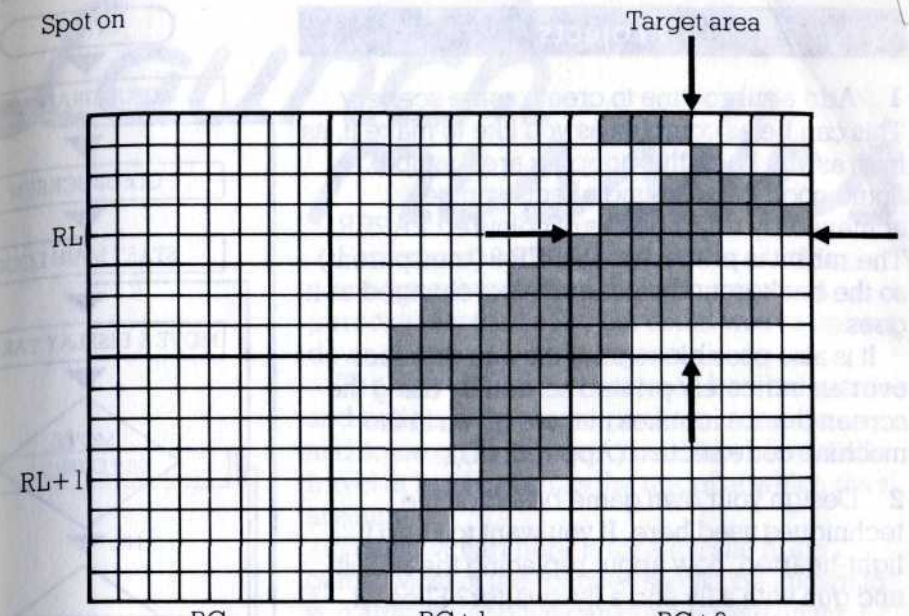
## Spot on

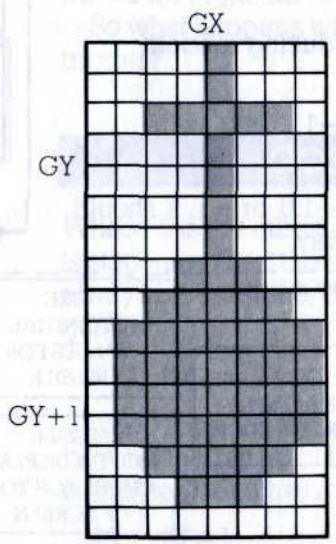
You want to make sure that the gunsight is directly over the target area on the rabbit. This is the line to do it:

```
IF GY=RL AND GX=RC+2 THEN....
```

The line checks the coordinates of the rabbit and the gun more closely than the original line does.





## Projects

**1** Add a subroutine to create some scenery. This can be as complex as you like to make it, as long as you keep the 'hopping area' simple. Some good background effects can be achieved by using blocks of coloured PAPER. The rabbit is printed on PAPER 8 (transparent) so the background colours are not changed as it goes.

It is also possible to print moving graphics over an intricately printed screen by using the screen dumps routines that are given in the machine code section (Appendix C).

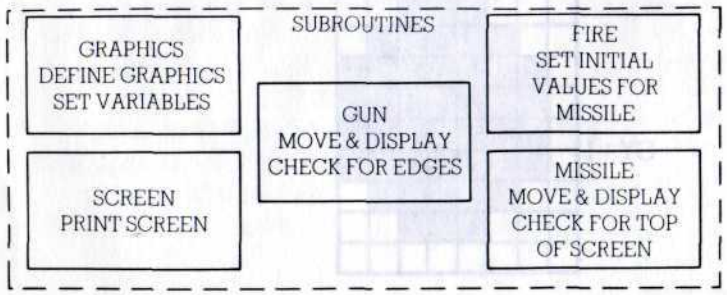
**2** Design your own game based on the techniques used here. If you want to keep it light-hearted, how about replacing the rabbit and gun with a fly and a fly-swatter? Flies, of course, don't hop, but they do buzz around unpredictably. This would change a variable by either +1 or −1:

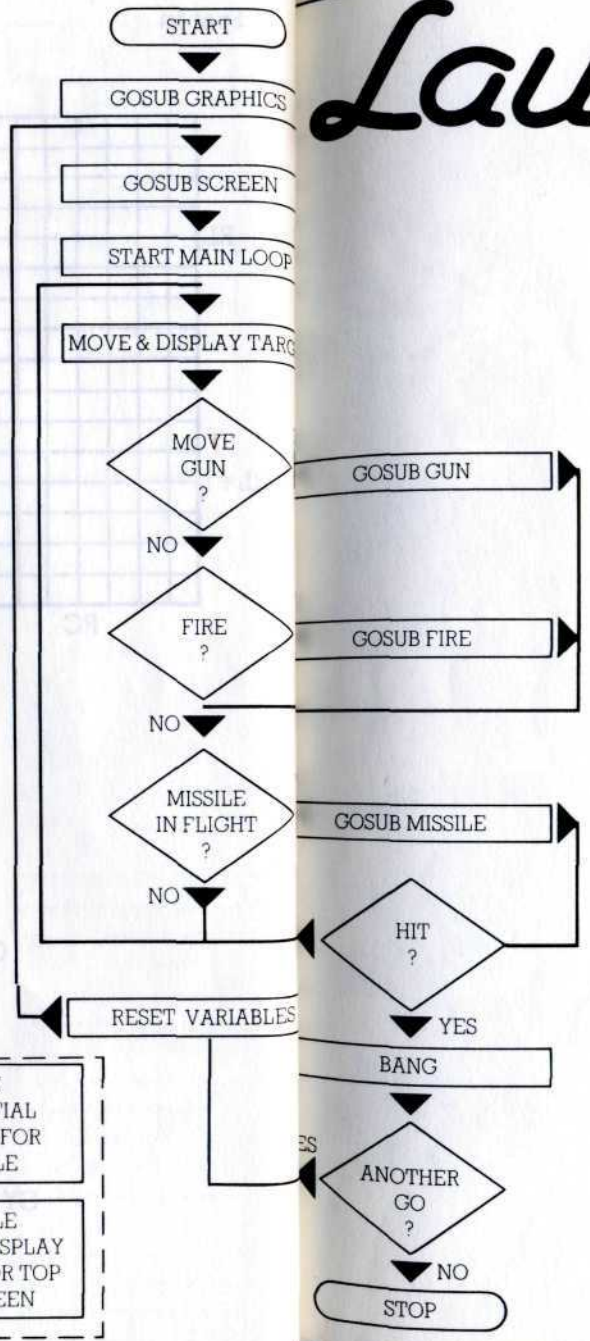
```
LET DX=1:IF RND>.5 THEN LET DX=-1
LET X=X+DX
```

Here's another way of producing random either-way movement:

```
LET DX=INT(RND*3)-1
LET X=X+DX
```

DX could have a value of +1,0, or −1. Y should be changed in the same way.






# Launch missiles

For a more warlike game, make a tank or plane your target. This should move more purposefully than a fly, but could weave up and down as it crossed the screen.

The only difference between bombs, missiles and bullets is that some go down, some go up, and some go in any direction. They should all travel at least as fast as the target at which they are aimed.

The flowchart here is for a missile firing game. Most of the routines should be known to you already, and we can concentrate on the parts of the program that handle the missile.

You will need two variables to handle the missile's position – MX and MY. You also need a flag to show whether or not a missile is in flight – MF=1 for flight, MF=0 for not yet fired.

So what happens when the player presses the trigger?

## GOSUB fire

Where has the missile started from? If your gun is stationary then the missile's X position will always be the same, otherwise find where the gun is:

```
LET MX = GX
```

The Y position is fixed. Set MY to a point just beyond the barrel of the gun. Now tell the computer that it has a missile to handle:

```
LET MF=1
```

## GOSUB missile

Is a missile in flight?

```
IF MF=1 THEN GOSUB.....
```

Move and display your target in the usual fashion, one step at a time for a slow missile. For a faster missile, then either move your missile in bigger jumps or use a loop. If you choose to use the Great Leap Forward method then watch out!

```
LET MY=MY-3
```
(the target moves 1 square at a time)

Your missile is now moving three times as fast as the target – but it's missing two out of every three positions. What happens if it misses the line that the target is on? This method will work, as long as you make sure that the leaps up the screen will bring the missile onto the target line. The Leap approach gives a fast movement, but the graphics suffer a little. The movement is jerky. You will get a smoother movement, and cover every possible line of the screen, by moving your missile through a loop. The catch here is that there will be a noticeable slowdown of the program – and of the target's speed, when the missile is in flight. This can be smoothed out by the addition of a line:

```
IF MF=Ø THEN FOR D=1 TO 25:NEXT D
```

The length of the delay needed will depend upon the rest of the program, but should be set so that the target's speed remains constant whatever the missiles are doing.



## Projects

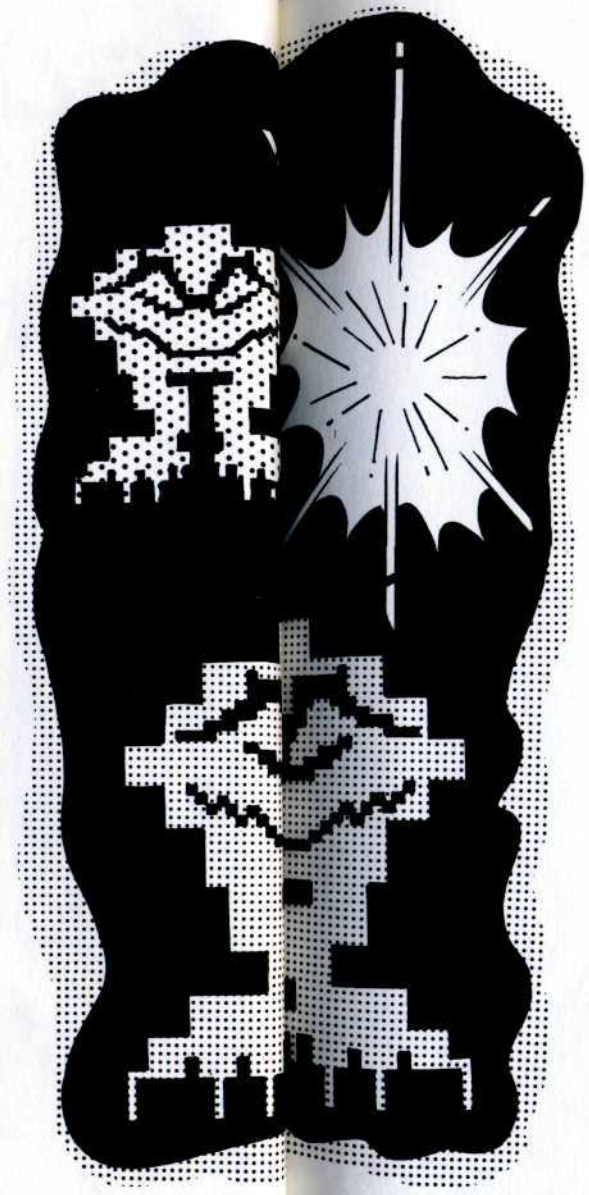
Turn the flowchart on page 41 into a game of your own. Use simple blob graphics and a blank screen at first, until you have got the routines working properly. You can also simplify the program initially by leaving the gun at a fixed place. The gun moving routine can be easily added later.

Use lots of imagination in your graphics, and in the game's background screen. Use even more in that BANG routine at the end. The speed and flow of the program no longer count at that point. You can have the target exploding, or crashing down in flames, with lots of sound effects.

Give your game a tighter structure by limiting the number of missiles, and keeping track of the scores. Instead of stopping and asking if the player wants another go, give him another directly, as long as he has missiles left. The missile count and hit count can be displayed on the screen during the game.

Turn the game on its head, and have the plane dropping bombs on a moving target. The bombs could travel diagonally, rather than vertically, to give a more interesting effect. An extra twist to this variation is to have the plane losing height all the time, so that the game has a very definite time limit. This would work best with multiple targets, for which, read on. . .

# String invaders



Here it is! A 15 line space invader program. It may not look much, but it's very impressive when you think how short it is. Type it in and run it to see what it does, and then we will look at how it does it.

```
10  LET I$=" X X X X X "
20  LET C=0: LET D=+1
30  CLS : PRINT AT 20,15; "↑"
40  PRINT AT 4,C;I$
50  LET A$=INKEY$: IF A$<> ""
    THEN GO SUB 100
60  LET C=C+D
70  IF C>=20 THEN LET D=-1
80  IF C<=0 THEN LET D=+1
90  PAUSE 5: GO TO 40
100 IF A$<>"1" THEN RETURN
110 FOR L=20 TO 5 STEP -1: PRINT
    AT L,15; " "
120 PRINT AT L-1,15;"↑": NEXT L
130 LET P=16-C
140 LET I$=I$( TO P-1) + " "
    +I$(P+1 TO )
150 PRINT AT 20,15; "↑": RETURN
```



## How it works

**1** The object of the game is to shoot the X's. Fire your missile, by pressing '1'.

**2** If you miss the string of invaders the program will crash.

**3** You have got all the time in the world. These invaders do not descend.

**4** The program doesn't check the condition of the invading fleet to see if it's wiped out, so break out of the program when you have shot them all.

## Strings

The key to the program is STRING SLICING – the art of chopping strings up and sticking them back together. You can cut individual characters out of string by telling the Spectrum the position of the character you want.

A$=[I][N][V][A][D][E][R][S]
A$(3)=[V]

To take a slice of several characters, give the positions of the first and last ones that are wanted.

A$(3 TO 7) = [V][A][D][E][R]

If the slice you want is on the left of the string, then you only need to give the position of the last character.

A$(TO 4) = [I][N][V][A]

Likewise, if you want a slice from the other end, just give the first position.

A$(5 TO ) = [D][E][R][S]

Strings can be joined together by using the addition sign. This is known as STRING CONCATENATION.

A$ = "SPACE" + "INVADERS" gives A$ = "SPACE INVADERS"

The main part of the program (30 to 90) shuffles the string of X's backwards and forwards across the screen. The D variable holds the direction of movement, and lines 70 and 80 turn it round at the limits of the movement.

The subroutine from 100 first whizzes the missile up the screen, then finds the place in the string (P) where the missile struck. The next line slices off the part of the string to the left of P and the part to the right, and rejoins them with a space in between. If there was an invader there, then there isn't now.

## Improvements

**1** Check for Hits. Did you get one? The way to answer that question is to find what was at place P in the string:

```
IF I$(P)="X" THEN LET HIT=HIT+1
```

**2** Add a Score. If you have a HIT counter, then this can be used as a basis for scoring. Write a PRINT "SCORE =";HITS*20 line into the end of that subroutine.

**3** Make them invade. Move the invaders steadily down the screen, dropping a line after every three or four passes. Keep a count of the number of times they have crossed, and hold their line position in a variable which is nudged on when COUNT = 4.
You can check to see if they have landed by looking at the line variable.

```
IF IL=20 THEN.....(end)
```

**4** Add some nice graphics – if you haven't already done so.

**5** Crashproof it. If you try to look at, or slice off, parts of the string which aren't there – that is, if the missile flies past the end of the string – then the program will crash. Prevent this by including a line to check that the value of P is within range.

**6** Shrinking Strings. Calculate the limits for the column position for printing, so that the string of invaders travels the full width of the screen.

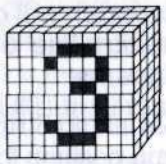
As invaders get shot at the ends of the string, the length of the string could be reduced. If the leftmost character is hit, then the string needs to be redefined to include only that section to its right:

```
IF P=2 THEN LET I$=I$(3 TO):
LET L=L-2
```

Turn this on its head to handle the other side.

```
IF P=L-1 THEN LET I$=I$( TO L-2):
LET L=L-2
```

The direction changing routine will need adjustment as well, now. The left limit for the print column (C) will always remain at 0, but the right limit will increase as the length of the string is reduced. You could get the computer to calculate this from the L variable, but the simplest method is to store the right limit in another variable, and alter that as the string is reduced.



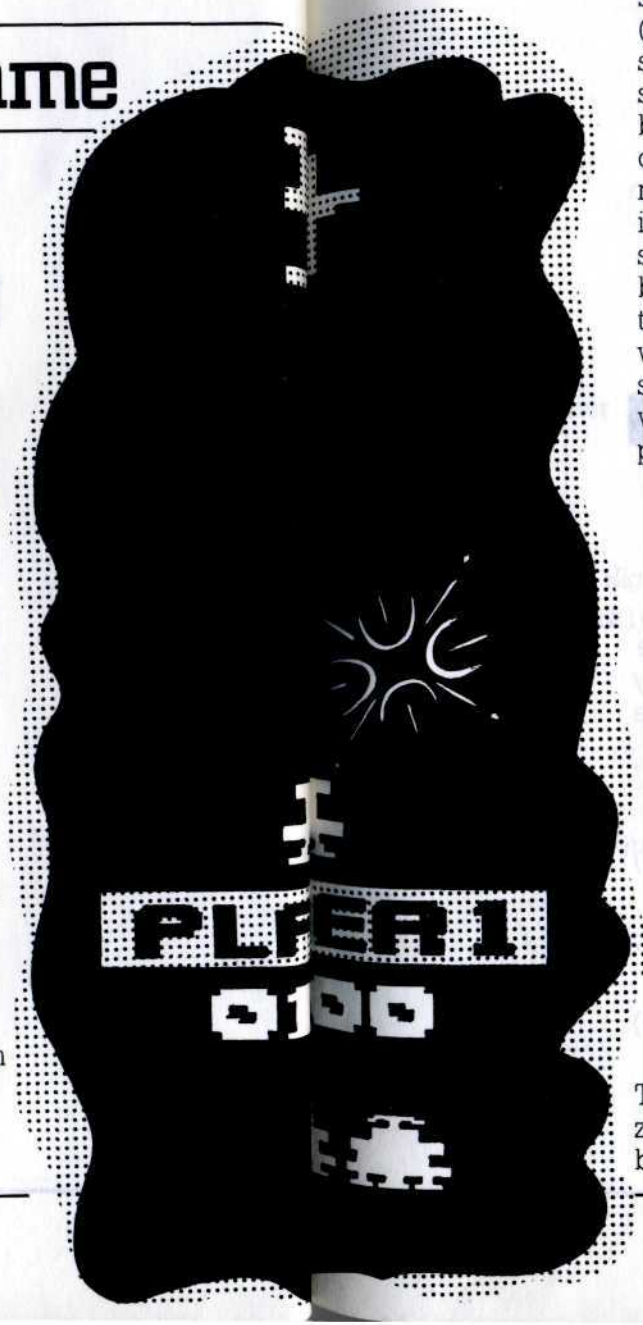
# The Hall of Fame

Every good game should have a Hall of Fame. How else will you keep track of the scores with so many of your friends and family queuing up to play?

To include a Hall of Fame in your program, you must store the names and scores of the best players, and the most convenient way to handle any sets of data is in ARRAYS. An array is a block of variables, all with the same name, but with different SUBSCRIPTS – reference numbers. S(5) would store five numbers in S(1), S(2), S(3), S(4) and S(5). All arrays start their numbering from 1. The size of the array is set, by you, early on in the program, with a line that gives the DIMENSIONS of the array.

Similarly, N$(5,10) would store five names. Here they would be limited to ten letters each, but they could be any length you choose, as long as you fix that length in the DIM line.

To use the routine, you need a couple of check lines at the end of the game. Is there an empty place still in the Hall of Fame? There will be at first, because the stores start off empty. If there's space, then go to the Hall routine. If not, then is the latest score better than the lowest score in the Hall? If it is, then it should be fitted into the Hall at the appropriate place. If not, then it's straight on to the next player.






## The routine

It would be possible to write a routine that compared the latest score with the recorded scores and then shuffled the lower scores down (and out) to make room for it. Possible, but not simple. The straightforward answer is to use a standard routine. The method given here works by reordering the set of numbers from the top down. It looks through the set to find the biggest number, and swaps this with whatever number is in the first store. It then looks at the list again, starting from the second store, and swaps the biggest unsorted number with the number in the second store. It moves on through the stores working the same way, until all have been shuffled into new places. This is a very efficient way of sorting lists. See for yourself. Type this program in, then enter six different numbers.

```
10  DIM N(6)
20  FOR T=1 TO 6
30  INPUT "NUMBER ";N(T)
40  NEXT T: LET P=0: GO SUB 200
50  FOR P=1 TO 5
60  LET H=-1
70  FOR T=P TO 6
80  IF N(T)>H THEN LET H=N(T):
    LET Z=T
90  NEXT T
100 LET X=N(P): LET N(P)=N(Z):
    LET N(Z) =X
110 GO SUB 200
120 NEXT P
130 STOP
200 FOR T=1 TO 6: PRINT AT P*3,
    T*4;N(T): NEXT T
210 RETURN
```

This routine only sorts positive numbers and zero. Some games produce negative scores – so beware.

## How it works

Look first at the set of lines from 60 to 100. H will store the highest number, but must be set to −1 before use. The T loop looks at each number in the array to see if this is higher than the highest number it knows. If it is, then the H variable is reset to a new high, and the reference number of the store is entered into Z. At the end of the loop it swaps the highest number (the one from N(Z)) with the one at the top of its list. The P loop moves the starting place for the T loop steadily downwards.

The print-out subroutine should show you what is happening, stage by stage.

## Linking in

With a few minor adjustments, this sorting routine can look after your Hall of Fame. The main change is to include the names in the swapping process. It's no good reordering the scores, and leaving the names the same way round. The arrays should be dimensioned to be one more than the number you intend to display – so that the latest score can be included. I have written it as a subroutine starting at 9000, and assumed a display of five names.

```
.....DIM N$(6,10):DIM S(6)
```
(include in initialization routine)

```
.....IF S(5)=0 THEN GOSUB 9000
```
(an empty place in the Hall)

```
.....IF SC>S(5) THEN GOSUB 9000
```
(better than lowest score)



```
9000  LET N$(6)=P$:LET S(6)=SC
9010  FOR P=1 TO 5
9020  LET H=-1
9030  FOR T=P TO 6
9040  IF S(T)>H THEN LET H=S(T):
      LET Z=T
9050  NEXT T
9060  LET X=S(P): LET S(P)=S(Z):
      LET S(Z)=X
9070  LET X$=N$(P):LET N$(P)=N$(Z):
      LET N$(Z)=X$
9080  NEXT P
9090  .....routine to print out
      Hall of Fame
```

FIRST PASS

| | Store | Number | |
|---|---|---|---|
| Start → | N(1) | 15 | |
| | N(2) | 37 | |
| | N(3) | 25 | |
| | N(4) | 71 | |
| | N(5) | 99 | ← Highest |

Swap N(5) and N(1)

SECOND PASS

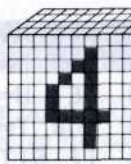
| | Store | Number | |
|---|---|---|---|
| | N(1) | 99 | |
| Start → | N(2) | 37 | |
| | N(3) | 25 | |
| | N(4) | 71 | ← Highest |
| | N(5) | 15 | |

Swap N(4) and N(2)

THIRD PASS

| | Store | Number | |
|---|---|---|---|
| | N(1) | 99 | |
| | N(2) | 71 | |
| Start → | N(3) | 25 | |
| | N(4) | 37 | ← Highest |
| | N(5) | 15 | |

Swap N(4) and N(3)

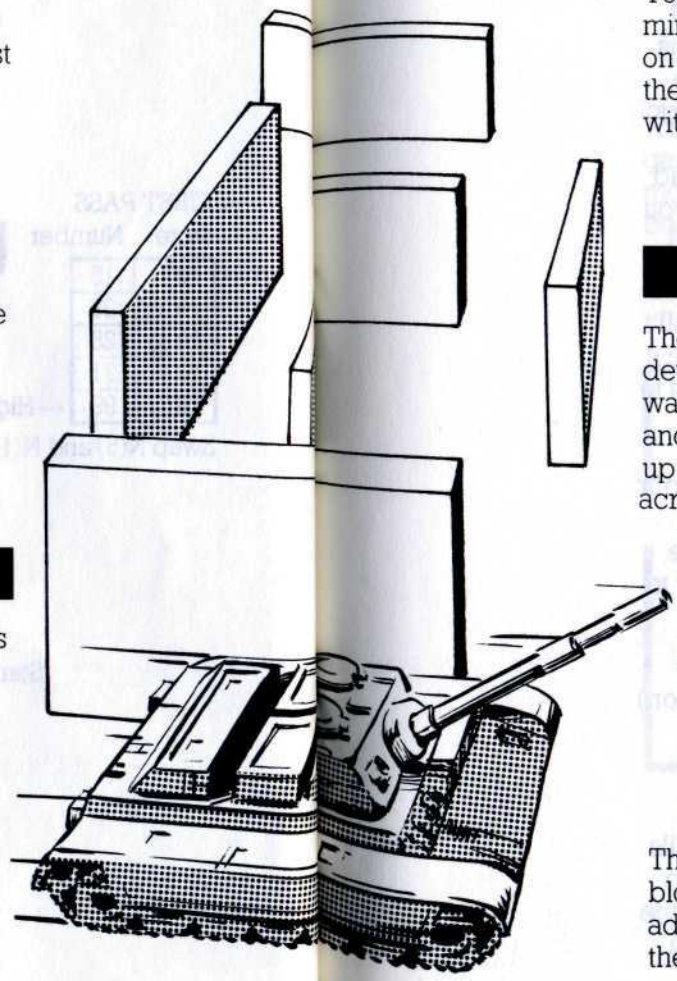


# Mazes

There are basically two different types of mazes, for two different types of games. The first type is a complex obstacle course, and is used in dodging, shooting and chasing games. The second type has confusing paths, and is used in exploring and adventure games. With either type of game, you have two alternative ways of creating the maze – either design it yourself, or get the computer to design it for you. Normally, the ones you design will look better, and can be more complicated than those produced by the machine, but your players will soon learn their way around them. Computer-generated mazes may not be as fine and fancy, but at least they will be random.

## Obstacle courses

The simplest way to set up an obstacle course is to scatter blocks around the playing area. The position of the blocks is shown on the screen, but is also recorded on a two-dimensional array. When the player tries to make a move, the coordinates of his intended position are checked against the array to see if the move is permissible. This is not actually necessary, as you could colour the blocks distinctively and check the ATTRibutes of the new position.





However, should you want a maze where the computer creates paths, then you would need to record it in an array. This will scatter 20 blocks on a 20×20 screen.

```
1000  FOR N=1 TO 20
1010  LET R=INT(RND*20)+1
1020  LET C=INT(RND*20)+1
1030  PRINT AT R,C;"■"
1040  NEXT N
```

You could use a similar system for scattering mines, or other traps. For this, put a character on the screen, to mark its position, but colour it the same as the screen. You can check for it with a line like this: (you have used 'M' for mine.)

```
IF SCREEN$(R,C)="M" THEN. . . . .
```

## Walls

These can be built at random, using a development of the block-scatter method. The walls need length, as well as a starting place, and some walls will go across the screen, others up and down. This produces random walls across the screen.

```
1000  FOR N=1 TO 10
1010  LET X=INT(RND*16)+1
1020  LET Y=INT(RND*20)+1
1030  LET Z=INT(RND*3)+2
1040  FOR Q=1 TO Z
1050  IF X+Q>20 THEN LET Q=Z:
      GOTO 1070
1060  PRINT AT Y,X+Q;"■"
1070  NEXT Q:NEXT N
```

These walls can be anything between 2 and 4 blocks long, (the Z variable). Notice the adjustments that have been made to ensure that the walls do not overrun the edge.

## Projects

**1** Try and write a tank game for two players. Get the computer to create the battlefield out of random horizontal and vertical walls. The routines needed for controlling two tanks are the same as for one, except that you will need to give each player his own set of key controls. Think the game through carefully before you start, and decide just how it is to be played. Will you allow the tanks to shoot their way through walls? What range will the tanks have? What is to happen after a tank has been hit? Does it carry on from where it was, but damaged, or go back to base? Do the tanks carry an endless supply of ammunition, or will they have to head for an arms dump every now and then? Will you give a tank the chance to escape when a shell has been fired at it?

**2** Design a chasing game. Use a random wall routine, or work out your own maze, and write it in as a print routine. Give the 'hero' something to find in the maze, and send some monsters in after him. You can create very aggressive monsters by making them head directly for your hero, wherever he may be. To do this, the program must compare the coordinates of the monster and the hero, and adjust the monster to bring it closer.

```
IF MX>PX THEN LET MX=MX-1
                    (PX=Player's X position)
IF MX<PX THEN LET MX=MX+1
IF MY>PY THEN LET MY=MY-1
IF MY>PY THEN LET MY=MY+1
```

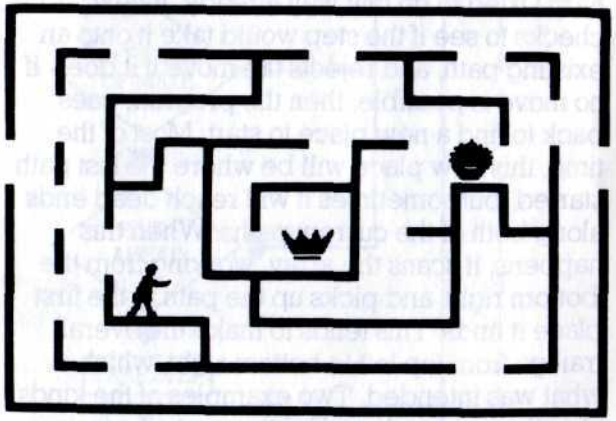
As the monster can no more walk through walls than can the hero, you have given the player a chance with this routine. It is possible to trap the monster by leading him into dead ends. If you think that this gives the hero too much of a



chance, and doesn't make the game sufficiently challenging, then add more monsters, all starting from different parts of the maze. The routine is exactly the same for ten monsters as for one, except that you no longer use simple variables. Hold the X and Y coordinates in an array, and run the monster-move routine through a loop.

```
FOR Q=1 TO 10
IF MX(Q)>PX THEN LET
MX(Q)=MX(Q)-1
....
etc
```

A simpler, but no less effective, way to make the game more exciting is to use ghosts, instead of monsters. Ghosts can walk through walls! If you do this, or if you have many monsters, then you really should give the player some means of defending himself. If you don't, then the games could be rather short.

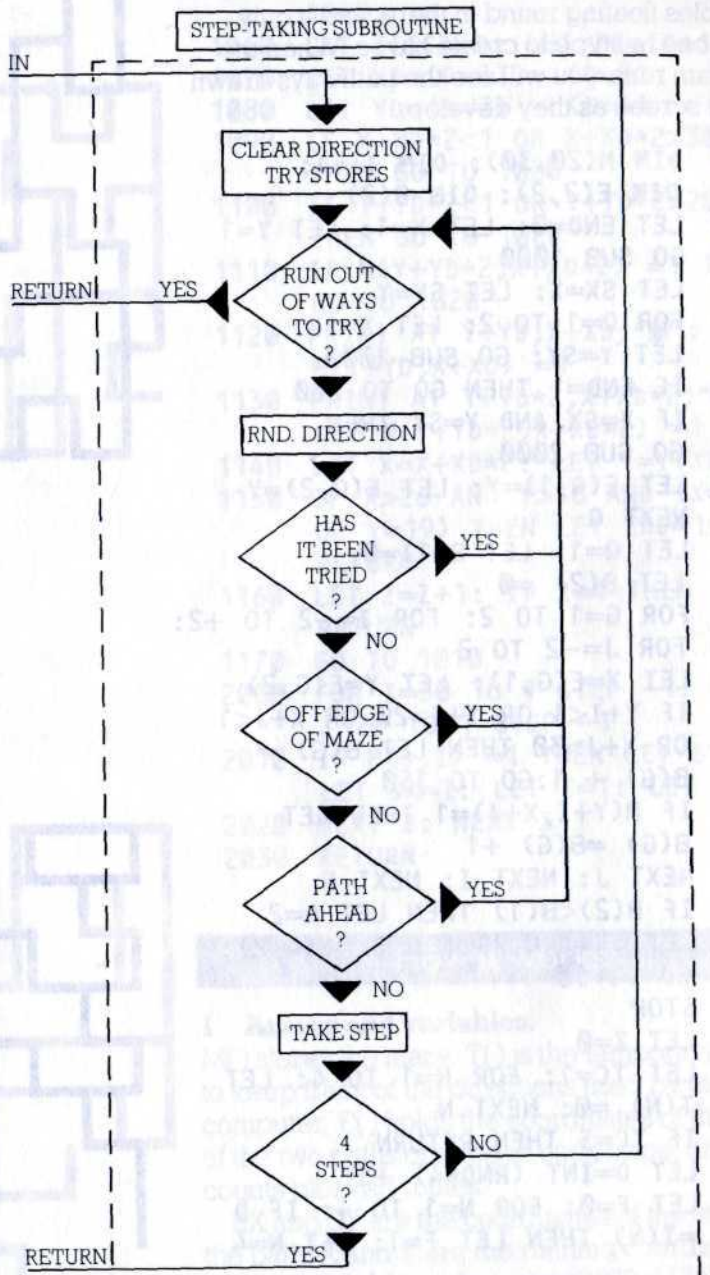


# Hidden paths

Here's a way to get the computer to generate a maze of confused and confusing paths. It works by laying a trail in stages, each of up to four steps. The steps cover two squares at a time so that it is spread out. At the end of a stage, it goes back to the point where it started that part of the path, and lays another path in a different direction. The program then looks at the two ends and sees which is in the most open position. That end becomes the start of the next stage – of the next pair of paths.

When the computer is taking a step, it picks a direction at random, and then checks that it hasn't tried to go that way already. It next checks to see if the step would take it onto an existing path, and rejects the move if it does. If no move is possible, then the program goes back to find a new place to start. Most of the time, this new place will be where the last path started, but sometimes it will reach dead ends along both of the current paths. When this happens, it scans the array, working from the 'bottom right' and picks up the path at the first place it finds. This tends to make the overall trail go from top left to bottom right, which is what was intended. Two examples of the kinds of pathways produced by this routine are shown. Look at them closely and you will see that although the paths wind backwards and forwards, they do not actually cross. This is important if you want to make sure that there is no way out of a dead end, except the way that you came.



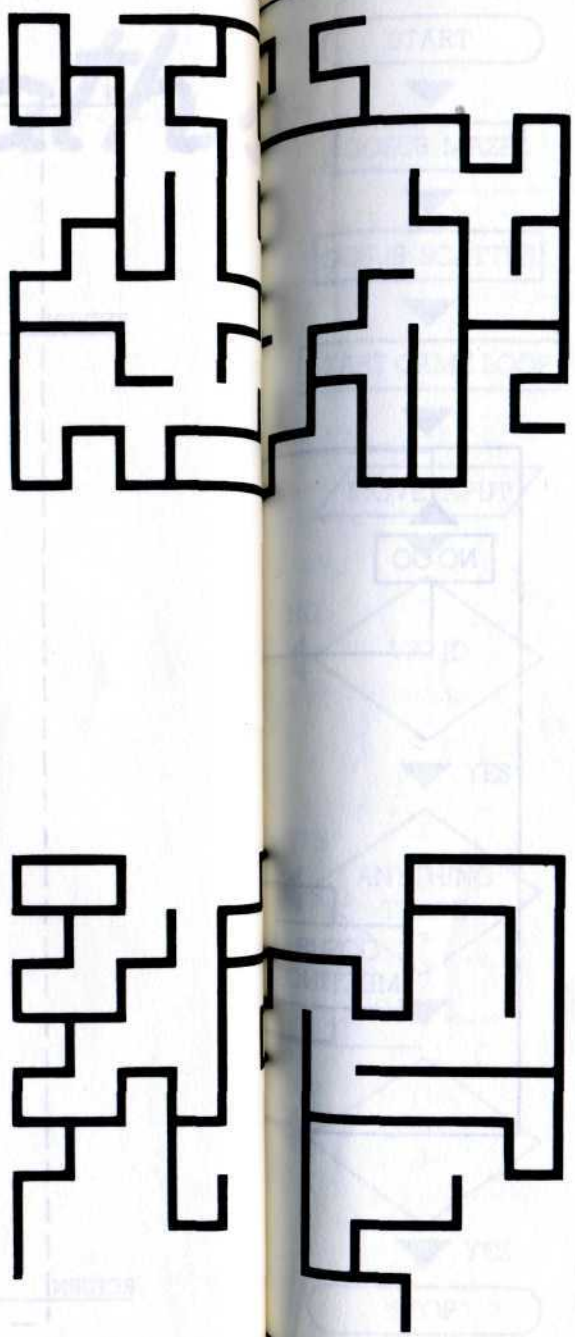





NADA PUBLISHING

Type the listing in carefully – there are a lot of variables floating round in there, and it only takes one mistype to create havoc. When the program runs, you will see the pathways drawn on the screen as they develop.

```
10  DIM M(20,30): DIM T(4):
    DIM E(2,2): DIM B(2)
20  LET END=0: LET X=1: LET Y=1
30  GO SUB 1000
40  LET SX=X: LET SY=Y
50  FOR Q=1 TO 2: LET X=SX:
    LET Y=SY: GO SUB 1000
60  IF END=1 THEN GO TO 200
70  IF X=SX AND Y=SY THEN
    GO SUB 2000
80  LET E(Q,1)=X: LET E(Q,2)=Y
90  NEXT Q
100 LET Q=1: LET B(1)=0:
    LET B(2) =0
110 FOR G=1 TO 2: FOR I=-2 TO +2:
    FOR J=-2 TO 2
120 LET X=E(G,1): LET Y=E(G,2)
130 IF Y+I<1 OR Y+I>20 OR X+J<1
    OR X+J>30 THEN LET B(G) =
    B(G) + 1:GO TO 150
140 IF M(Y+I,X+J)=1 THEN LET
    B(G) =B(G) +1
150 NEXT J: NEXT I: NEXT G
160 IF B(2)<B(1) THEN LET Q=2
170 LET SX=E(Q,1):LET SY=E(Q,2):
    GO TO 50
200 STOP
1000 LET Z=0
1010 LET TC=1: FOR N=1 TO 4: LET
    T(N) =0: NEXT N
1020 IF TC=5 THEN RETURN
1030 LET D=INT (RND*4) +1
1040 LET F=0: FOR N=1 TO 4: IF D
    =T(N) THEN LET F=1: LET N=4
```





```
1050 NEXT N: IF F=1 THEN GO TO
    1020
1060 LET T(TC)=D: LET TC=TC+1
1070 LET XD=(D=1) - (D=3)
1080 LET YD=(D=2) - (D=4)
1090 IF X+XD*2<1 OR X+XD*2>30
    THEN GO TO 1020
1100 IF Y+YD*2<1 OR Y+YD*2>20
    THEN GO TO 1020
1110 IF M(Y+YD*2,X+XD*2) =1 THEN
    GO TO 1020
1120 PRINT AT Y+YD,X+XD;"■": LET
    M(Y+YD,X+XD) =1
1130 PRINT AT Y+YD*2,X+XD*2;"■":
    LET M(Y+YD*2,X+XD*2) =1
1140 LET X=X+XD*2: LET Y=Y+YD*2
1150 IF X>26 AND Y>16 AND (X=29
    OR Y=19) THEN LET END=1:
    RETURN
1160 LET Z=Z+1: IF Z=4 THEN
    RETURN
1170 GO TO 1010
2000 FOR I=30 TO 1 STEP -1: FOR
    J=20 TO 1 STEP -1
2010 IF M(J,I) =1 THEN LET SY=J:
    LET SX=I: LET I=1: LET J=1
2020 NEXT J: NEXT I
2030 RETURN
```

## How it works

**1 Arrays and variables.**
M() stores the maze, T() is the temporary store to keep track of the directions tried by the computer, E() holds the coordinates of the ends of the two paths created at each stage, and B() counts blocked routes.

SX and SY are the coordinates of the start of the path, X and Y are the running coordinates. In the step-taking subroutine (1000–1170), TC is

the try counter, Z counts the steps, and XD and YD are the change of position variables. (One of these will be 0, the other will be −1 or +1.)

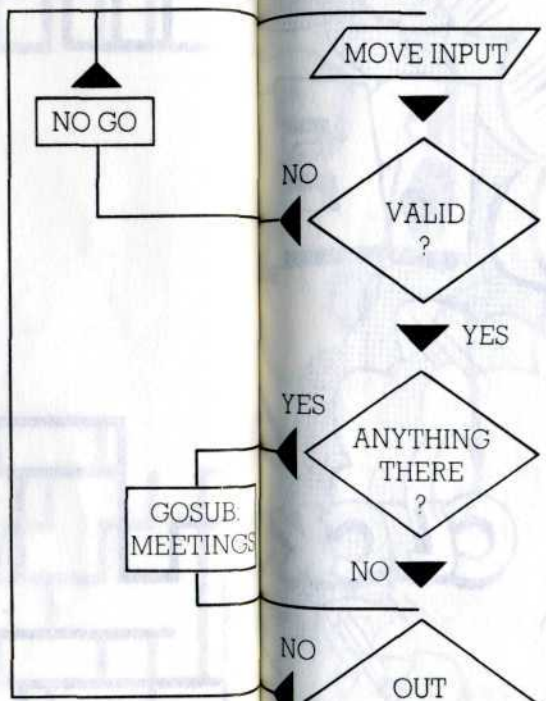
**2 The subroutine at 1000.** The computer picks a direction (1030), checks that it hasn't tried to go this way before (1040) and, all being well, stores the direction for future reference (1060). The direction is then converted into a form suitable for coordinate work (1070–1080). It then checks that the move will remain in the maze area, and that it will not join up with an existing path (1090–1110), and then takes the step – or rather, two steps – in the same direction. Finally it updates the X and Y values and checks to see whether it has reached a suitable place on the edge, and whether it should take another step at that stage.

**3 The main routine.** The first steps taken by the program are unusual in that the subroutine is only visited once at this point. This ensures that the path gets into the maze before it starts to divide. Lines 50 to 90 then take it through the subroutine twice, starting from the same place each time, and storing each endpoint. If no steps were taken by the subroutine (line 70), then the program goes to 2000 to find another place to start the path.

The lines from 100 to 150 assess mobility. How many of the possible steps around the end of each path are blocked by the edges of the maze, or by existing paths? The end which has most openings becomes the start of the next stage. If you wish to speed up this program, then this whole routine can be replaced by a simple random switch.

```
LET Q=1: IF RND > .5 THEN LET Q=2
```

The program will lose a little finesse, but the end result will not be too different!









## Project

Use a maze-making routine to create an exploration game. The basic flowchart for such a game is given here. First get the computer to generate your maze, but do not display it. Next scatter a variety of happenings about the pathways. Some of these should be good things to find, like bags of gold or princesses in need of rescue (they should be worth lots of points!); other things will be not so good – evil goblins, fire-breathing dragons or whatever. Make sure that these are placed on the paths by checking their coordinates against the maze array and rejecting any not on paths. Enter them into the array in code. 1 indicates path, so 2 could show the presence of gold, 3 dragons, etc.

You will then need to add a routine to allow the player to move about the maze. You might like to display his movements, or, if you want to give your players a real challenge, you could continue to leave the screen blank. As he tries to make a move, check that there is a path there, and look to see if he is about to meet something interesting. All of the meetings will be dealt with by separate subroutines. Quite how you deal with a fire-breathing dragon in a maze is a problem for you to solve! The last check is to see whether or not he has reached the end. When he does, you can give a final score based on the quantity of treasure, and the number of wounds he has acquired in his travels.

Exploration games of this type are sometimes referred to as graphic adventures, and they have much in common with adventure games, of which more in the next section.

# Special effects

## Basic beeps

Spectrum's BEEP command is very easy to use, but rather limited in the range of sounds that it can produce. Running a series of short BEEPs through a loop gives some interesting effects. Try this:

```
10  FOR N=0 TO 40 STEP 5
20  BEEP .005,N
30  NEXT N
40  GOTO 10
```

There are lots of variations on this idea. The length of the note depends on the first number after the BEEP command. Below .002, the sound is a simple click, with the pitch almost completely lost. Altering the size of the step changes the jumps in pitch. Add another line to give an undercurrent to the sound:

```
25  BEEP .005,0
```

The pitch of the note in this line could also be related to the value of N. BEEP .005,N–10 sounds quite interesting.






## Space sounds

The range of sound effects you can get from the Spectrum can be greatly increased by the use of a simple machine code routine. The one given here will produce a weird and wonderful variety of sounds by putting in different values at the places marked in the DATA lines. Use this program to explore the possibilities, and write down any combinations that you find particularly pleasing.

```
10  CLEAR 32499: LET M=32500
20  FOR T=0 TO 25: READ B
30  POKE M+T,B: NEXT T
40  DATA 6,5:REM 5= DURATION
    (0 TO 255)
50  DATA 197,33,0,5:REM 5=
    FREQUENCY (0 TO...)
60  DATA 17,1:REM 1=TONE (0 TO
    255)
70  DATA 0,229,205,181,3,225
80  DATA 17,16: REM 16 = INTERVAL
    (1 TO...)
90  DATA 0,167,237,82,32,240,
    193,16,233,201
100 RANDOMIZE USR M
110 PAUSE 0
120 GOTO 100
```

The top limits to the FREQUENCY and INTERVAL values have not been given, as this is for you to explore, but neither should be too high, or you will find that the routine takes an amazing time to run.

Some combinations will give you siren effects, others could well be used for lasers.

## Lasers

If you want laser sounds, then you will be needing laser graphics as well. Here are a couple of ideas to try. Both use high-resolution graphics, though in different ways.

```
1000  OVER 1: FOR N=1 TO 2
1010  PLOT X,Y: DRAW X1, Y1
1020  NEXT N:RETURN
```

This is really too fast to be of much use as it stands, but by adding a command to call up your machine-code sound effect, you will slow it to a more reasonable speed.

A series of plots takes longer to appear.

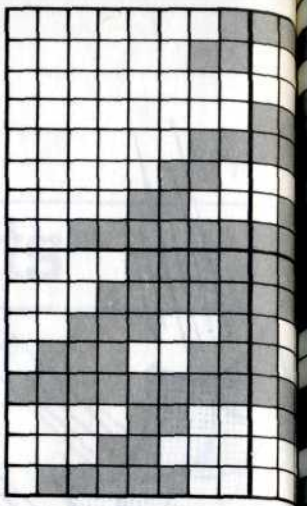
```
1000  OVER 1: FOR N=1 TO 2
1010  FOR P=X TO X1
1020  PLOT P,Y: RANDOMIZE USR M
                          (sound effect)
1030  NEXT P
1040  NEXT N
```

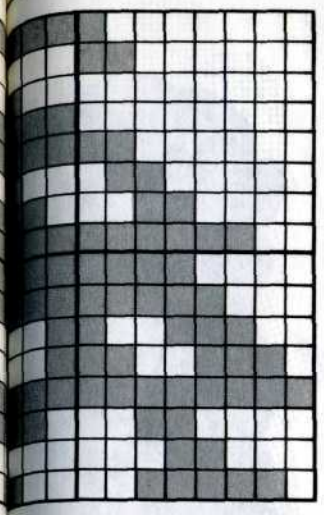
In both of these routines, X,Y is the position of the gun and X1,Y1 is the target. As both of these will probably be printed on screen, using line and column coordinates, you will need to convert the print coordinates to high-res. coordinates. Use these formulae:

```
X= C*8 + 4
Y= (21 - L)*8 + 4
```

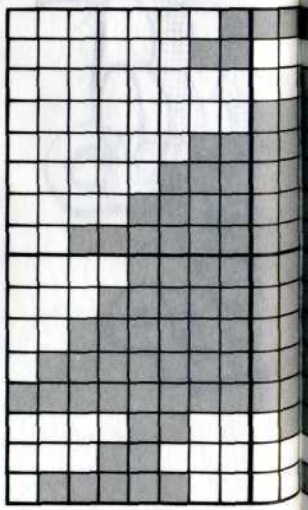
If you need to convert back to print coordinates, then use these:

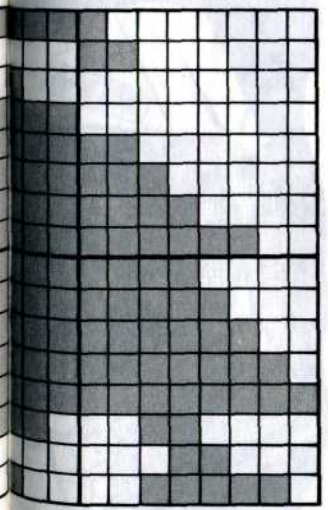
```
C= INT (X/8)
L= 21 - INT(Y/8)
```





ABC and DEF are the user-defined graphics (UDG) for the first flying saucer and GHI and JKL are those for the second.





## Spinning saucers

It is a well-known fact that all flying saucers have flashing navigation lights. Have yours got them? You can make the whole saucer flash by using the FLASH control, but you may want a more subtle effect than this. Here's one way to do it. It uses two separate images of the saucer, one with lights on, and the other with them off. Two images are held in an array, and printed alternately.

```
1000  FOR N=1 TO 12: READ G$
1010  FOR R=0 TO 7: READ B
1020  POKE USR G$+R,B
1030  NEXT R: NEXT N
1040  DATA "A",1,3,0,0,3,6,12,63
1050  DATA "B",255,24,60,255,255,60,
      126,255
1060  DATA "C",128,192,0,0,192,96,
      48,252
1070  DATA "D",15,31,57,115,255,12,
      24,124
1080  DATA "E",255,255,153,153,255,
      126,24,60
1090  DATA "F",240,248,156,206,255,
      48,24,62
1100  DATA "G",1,3,0,0,3,7,15,63
1110  DATA "H",255,24,60,255,255,
      255,255,255
1120  DATA "I",128,192,0,0,192,224,
      240,252
1130  DATA "J",15,31,63,127,255,12,
      24,124
1140  DATA "K",255,255,255,255,255,
      126,24,60
1150  DATA "L",240,248,252,254,255,
      48,24,62
1160  DIM A$(2,2,3)
1170  FOR T=1 TO 2: FOR N=1 TO 2
1180  READ A$(T,N): NEXT N: NEXT T
```

```
1190  DATA "ABC",DEF","GHI","JKL"
1200  PRINT AT 10,10;A$(1,1); AT
      11,10;A$(1,2)
1210  PAUSE 10
1220  PRINT AT 10,10;A$(2,1); AT
      11,10;A$(2,2)
1230  PAUSE 10: GOTO 1200
```

This gives a steady flash. To turn this into a spinning effect, you need to make a few adjustments. Instead of two images, you will need a set, with different lights on in each image. They can then be printed one after the other, so that it looks as if there is a single light on a rotating saucer.

## Speed up the action

There will probably come a time when you feel that you simply cannot get the speed you want by writing in BASIC, and you will start to look round for alternatives. Machine coding is an obvious route to take, but not an easy one. Even experienced machine code programmers find that it takes days or weeks to produce routines that do the same as BASIC routines that were written in a few hours. There are less taxing alternatives that give results that are almost as good.

## Compiler programs

BASIC is slow because the computer has to interpret every command as it goes through the program. When it meets an instruction, it has to work out what it means, call up the appropriate routine from ROM, look up any variables or coordinates that are being used and then implement it. The computer also has to sort




through the program every time it meets a GOTO or GOSUB instruction, and count its way down to the appropriate line. It all takes time. A COMPILER PROGRAM will take your BASIC program and turn it into machine code, eliminating all of this interpretation and line finding. The result is a much faster program. There are catches though. Compilers won't handle any string arrays, some won't handle any arrays at all, or any strings, and most are protected by copyright. If you wrote a game which you hoped to sell, you could not use someone else's compiler to give you the speed you needed.

## Special languages

There are several versions of FORTH on the market at the moment. This language is far more structured than BASIC, and runs much quicker. It would be ideal for games writing, except that again, it is protected by copyright, and to run a FORTH program, you need to have the language in the micro already. It's suitable for enthusiasts who will simply swap games around between themselves, but of little value if you are hoping to sell your game on the open market.

SCOPE (from ISP Marketing) offers a better solution. SCOPE routines are not unlike BASIC to write, and the complete routine can be compiled into a block of independent machine code which will work without the SCOPE language being present. This can be a complete program, or a set of routines to call up from a BASIC program. The SCOPE code can be saved in the same way that any machine code can be, and there is no copyright problem about using SCOPE in games that you hope to sell.

# Program listings




```
 10 REM "ORBIT"
 15 REM
 20 REM SHOOTING AND BOMBING
 25 REM
 30 LET G=0: LET H=0: RESTORE 6000:
    GO SUB 6000
 40 PRINT AT 21,0;" *** READY -
    PRESS ANY KEY ***": PAUSE 0
100 BORDER 4: PAPER 6: INK 0: CLS
110 INK 1: PRINT AT 14,14;"■";
    AT 15,6;"■";AT 15,13;"■■■";
    AT 16,6;"■■";AT 16,12;
    "■■■■■";AT 16,26;"■■■"
120 PRINT AT 17,2;"■■";AT 17,6;
    "■■";
    AT 17,11;"■■■■■■■";
    AT 17,26;"■■■■■"
130 PRINT AT 18,2;"■■";AT 18,6;
    "■■";AT 18,10;"■■■■■■■■■";
    AT 18,26;"■■■■■"
140 FOR L=10 TO 18: PRINT AT L,21;"■■■": NEXT L
150 FOR L=19 TO 21: PRINT INK 3;
    AT L,0;"■■■■■■■■■■■■■■■■
    ■■■■■■■■■■■■■■■■■■■■":
    NEXT L
```



```
199 REM
200 REM STARTING VARIABLES
210 LET L=6: REM SHIP LINE
220 LET C=0: REM SHIP COLUMN
240 LET E=0: REM NO ENEMY
    FIGHTERS - YET
250 LET B=0: REM NOT DROPPING
    BOMBS
260 LET F=0: REM LASERS NOT
    FIRING
270 LET SN=5: REM NUMBER OF
    SPACESHIPS
290 REM
300 REM MAIN LOOP
305 REM GRAPHICS IN 310 AND 320
310 FOR N=1 TO SN: PRINT AT
    20,N*2; INK 0;"A": NEXT N
320 INK 0: PRINT AT L,C;"A"
340 LET A$=INKEY$
345 REM
346 REM DROP BOMBS
347 REM
350 IF B=0 AND A$="0" THEN LET
    BL=L: LET BC=C: LET B=1
355 REM
356 REM FIRE LASERS
357 REM
360 IF A$="1" THEN GO SUB 3000
370 PRINT AT L,C;" "
380 LET L=L+(A$="6")*(L<18)-
    (A$="7")*(L>0)
390 LET C=C+1-(32 AND C>30)
410 IF F=0 AND RND>.9 THEN LET
    F=1: LET G=INT (RND*10):
    LET H=31
415 REM
416 REM CRASH INTO BUILDING?
417 REM
420 IF ATTR (L,C)=49 THEN
    GO TO 600
```

```
430  OVER 0: INK 0: PRINT AT
     L,C;"A"
440  IF B THEN GO SUB 2000
450  IF F THEN GO SUB 1000: REM
     FIGHTER?
460  IF F=-1 THEN GO TO 600
470  GO TO 340
600  FOR N=1 TO 6: PRINT AT L,C;
     INK N;"A": BEEP .01,N*4:
     NEXT N
610  PRINT AT L,C;" "
620  PRINT AT 20,0; INK 3;
     "■■■■■■■■■■■■■■"
630  LET SN=SN-1: IF SN=0 THEN
     GO TO 700
640  IF F=0 THEN GO TO 670
650  FOR N=H TO 0 STEP -1: IF
     ATTR (G,N)=49 THEN GO SUB
     3100: GO TO 670
655  REM
656  REM G,H ARE LINE & COLUMN
     VARIABLES OF ALIENS
657  REM
660  PRINT AT G,N;"C": BEEP
     .02,25: PRINT AT G,N;" ":
     NEXT N
670  LET F=0: LET L=6: LET C=0:
     GO TO 300
700  STOP
1000 REM
1001 REM MOVE ALIENS
1002 REM
1005 PRINT AT G,H;" ": LET
     H=H-1+(32 AND H<1): LET
     G=G+(L>G)-(L<G)
1007 REM
1008 REM CRASH INTO BUILDING?
1009 REM
1010 IF ATTR (G,H)=49 THEN GO TO
     3100
1020 PRINT AT G,H; INK 2;"C"
```

Graphics occur in the following lines:
430, 600, 660
1020, 2030, 3100



```
1027 REM
1028 REM CRASH INTO SHIP?
1029 REM
1030 IF G=L AND H-C<=1 THEN LET
     F=-1
1070 RETURN
2000 REM
2001 REM BOMB ROUTINE
2002 REM
2005 PRINT AT BL,BC;" ": LET
     BL=BL+1
2010 IF ATTR (BL,BC)>48 THEN GO
     TO 2100
2030 PRINT AT BL,BC;"B"
2040 RETURN
2100 IF BL=19 THEN GO TO 2200
2110 FOR T=1 TO 3: PRINT AT
     BL,BC;"*": BEEP .01,10: PRINT
     AT BL,BC;" ": NEXT T
2200 LET B=0: RETURN
3000 REM
3001 REM LASERS
3002 REM
3005 IF C>24 THEN RETURN
3010 INK 3: OVER 1
3020 PLOT FN C(),FN L(): DRAW
     48,0
3040 OVER 1: PLOT FN C(),FN L():
     DRAW 48,0
3050 OVER 0: IF ATTR (G,H)<>51
     THEN RETURN
3100 LET F=0: FOR N=3 TO 6:
     PRINT AT G,H; INK N;"C":
     BEEP .02,N*5: NEXT N
3110 PRINT AT G,H;" ": RETURN
4999 STOP
6000 PAPER 7: INK 0: CLS
6010 PRINT AT 2,13;"ORBIT"
```

```
6020  PRINT AT 5,2;"YOU ARE IN
      ORBIT AROUND THE ";AT 7,2;
      "PLANET XERON. YOUR TASK IS
      TO"
6030  PRINT AT 9,2;"FLATTEN THE
      BUILDINGS, AND TO";AT 11,2;
      "BEAT OFF ENEMY FIGHTERS."
6040  PRINT AT 14,2;"CONTROLS:";AT
      16,3;"6 UP 7 DOWN";AT 18,3;
      "1 FIRE LASERS 0 DROP BOMBS"
6100  REM GRAPHICS
6110  FOR N=1 TO 3: READ G$: REM
      WHICH GRAPHIC LETTER
6120  FOR R=0 TO 7: READ B: REM
      NUMBER FOR DEFINING EACH ROW
6130  POKE USR G$+R,B: NEXT R:
      NEXT N
6140  REM SPACESHIP
6150  DATA "A",0,224,124,50,255,
      124,0,0
6160  REM BOMB
6170  DATA "B",48,48,208,252,30,
      30,14,0
6180  REM ALIEN FIGHTER
6190  DATA "C",0,60,118,173,247,
      126,36,66
6200  REM FUNCTION DEFINITION
6230  DEF FN L()=171-L*8
6240  DEF FN C()=C*8+8
6250  RETURN
```



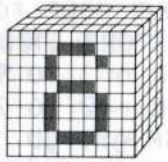
# FIREFIGHT

```
 10  REM "FIREFIGHT1"
 20  GO SUB 5000: REM front page
     and graphics
 30  GO SUB 6000: REM screen
     display and variables
 36  REM main loop starts here
 40  IF INKEY$<>"" THEN GO TO 40
 50  PRINT AT l,c;e$(d)
 60  IF PEEK 23672>tl THEN GO SUB
     1000
 68  REM fires out of control
 70  IF f>25 THEN GO TO 500
 80  LET a$=INKEY$
 88  REM quit
 90  IF a$="Q" THEN GO TO 600
 98  REM squirt
100  IF a$="1" THEN GO SUB 1100:
     GO TO 40
120  IF a$<"5" OR a$>"8" THEN
     GO TO 50
128  REM direction change
130  LET d=d+(a$="8")-(a$="5"):
     LET d=d+(4 AND d<1)-(4 AND
     d>4)
140  PRINT AT l,c;"  "
148  REM movement
150  IF a$<>"7" THEN GO TO 40
160  LET l1=l-(d=4)*(l>1)+
     (d=2)*(l<20)
170  LET c1=c-(d=3)*(c>1)+
     (d=1)*(c<29)
180  IF s$(l1,c1 TO c1+1)<>"  "
     THEN GO TO 50
190  LET l=l1: LET c=c1
```

```
 200 GO TO 50
 500 BEEP 2,10: PRINT AT 0,5;
     "ENOUGH!!!": STOP
 600 STOP
 998 REM put arsonist to work
1000 LET fl=INT (RND*20)+1: LET
     fc=INT (RND*29)+1: IF ABS
     (fl-l)<5 AND ABS (fc-c)<5
     THEN GO TO 1000
1010 LET s$(fl,fc)="f": PRINT AT
     fl,fc; PAPER 6; INK 2; FLASH
     1;"I": LET f=f+1: REM GRAPHICS
1020 POKE 23672,0: RETURN
1098 REM find position and
     direction of hose
1100 LET fl=l-(d=4)*(l>1)+
     (d=2)*(l<20)
1110 LET fc=c-(d=3)*(c>1)+
     (2 AND d=1)*(c<29)
1118 REM check for fire
1120 IF s$(fl,fc)<>"f" AND
     s$(fl,fc+1)<>"f" THEN RETURN
1130 LET s$(fl,fc TO fc+1)="  ":
     PRINT AT fl,fc;"  ": REM TWO SPACES
1140 LET tl=tl-5: REM reduce time
     limit
1150 LET f=f-1: REM 1 less fire
1160 RETURN
5000 PRINT AT 1,10; PAPER 6; INK
     2; FLASH 1; " FIREFIGHT "
5010 PRINT AT 3,2;"Steer the fire
     engine around";AT 5,2;"the
     screen, putting out fires."
5020 PRINT AT 7,2;"You lose if
     the fires get out";AT 9,2;
     "of control!"
5030 PRINT AT 12,2;"Use these
     keys:";AT 14,3;"STEER RIGHT 8
     LEFT 5";AT 16,9;"GO FORWARD
     7";AT 18,3;"SQUIRT 1 QUIT Q"
5050 PRINT AT 21,0;"HANG ONTO
```



```
     YOUR POLE FOR A MINUTE"
5100 RESTORE 5100
5110 FOR n=1 TO 9: READ g$
5120 FOR r=0 TO 7: READ b: POKE
     USR g$+r,b
5130 NEXT r: NEXT n
5140 DATA "a",0,7,125,255,255,255,
     60,24,"b",127,220,242,241,
     255,255,60,24: REM right
5150 DATA "c",3,7,9,9,15,12,31,4,
     "d",128,192,32,32,224,96,
     240,64
5160 DATA "e",254,59,79,143,255,
     255,60,24,"f",0,224,190,255,
     255,255,60,24
5170 DATA "g",3,4,15,12,15,15,31,4,
     "h",128,64,224,96,224,224,240,64
5180 DATA "i",16,16,16,25,187,191,126,60
5200 DIM E$(4,2): FOR N=1 TO 4:
     READ E$(N): NEXT N
5210 DATA "AB","CD","EF","GH": REM GRAPHICS
5220 LET TL=200: REM TIME LIMIT
5380 PRINT AT 21,0; "PROGRAM
     READY - ANY KEY TO START"
5390 BEEP 1,0: PAUSE 0
5400 RETURN
6000 DIM s$(20,30): BORDER 4:
     PAPER 7: INK 0: CLS
6010 PRINT PAPER 2;"  (32 spaces)  "
6020 FOR n=1 TO 20: PRINT AT
     n,0; PAPER 2;" ";AT n,31;" ":
     NEXT n
6030 PRINT AT 21,0; PAPER 2;" (32 spaces) "
6040 FOR N=1 TO 25: LET L=INT
     (RND*10)*2+1: LET C=INT
     (RND*15)*2+1
6050 LET s$(l,c)="1": PRINT AT
     l,c; PAPER 0;" ": NEXT N
6060 LET l=1: LET c=1
6070 LET d=1: LET f=0
6080 RETURN
```



# The spirit of adventure

Adventure games should challenge the players' ingenuity, wit and staying power. Cunning, not speed, is the key to success in these games, and this means that the speed at which the program runs is not particularly important either. That's good news, because it means that you do not have to think about machine code, or about making sure that your program is always written in the most efficient and fastest way.

What is an Adventure Game? You have probably played them already, but just in case you haven't met them before . . .

An Adventure Game is like an adventure story, except that you – the player – are part of that story. It will usually be set in some exotic place – a Lost Temple, a mad scientist's secret base, the Wild West, an alien planet, the ocean floor, a jungle, or whatever. The object of the game is usually to explore all the parts of this place, meeting and overcoming terrible dangers, and eventually finding some wonderful object, treasure, captured princess, magic wand, or even the Lost Crown of the Umbamajini!

The game is played as a series of events, with the computer first telling the player which 'room' that he is in, and something about it – what he can see there, and which ways he can leave. The player then tells the computer what he wants to do. The instruction might be to go to





the next room, or to pick up an object that he can see, or what he is going to do to stop the 'great hairy monster with gnashing teeth and rolling eyes' from eating him up. These instructions are given using a few simple words, which the computer has been programmed to understand.

Some adventure games go beyond this, and are more a variety of 'Dungeons and Dragons', where the player is one of a band of explorers, the others being controlled by the computer. Each player has his own special mixture of strengths and abilities – he may be a magician, a warrior, a burglar, a dwarf. The player's character is decided at the beginning of the game by a random process.

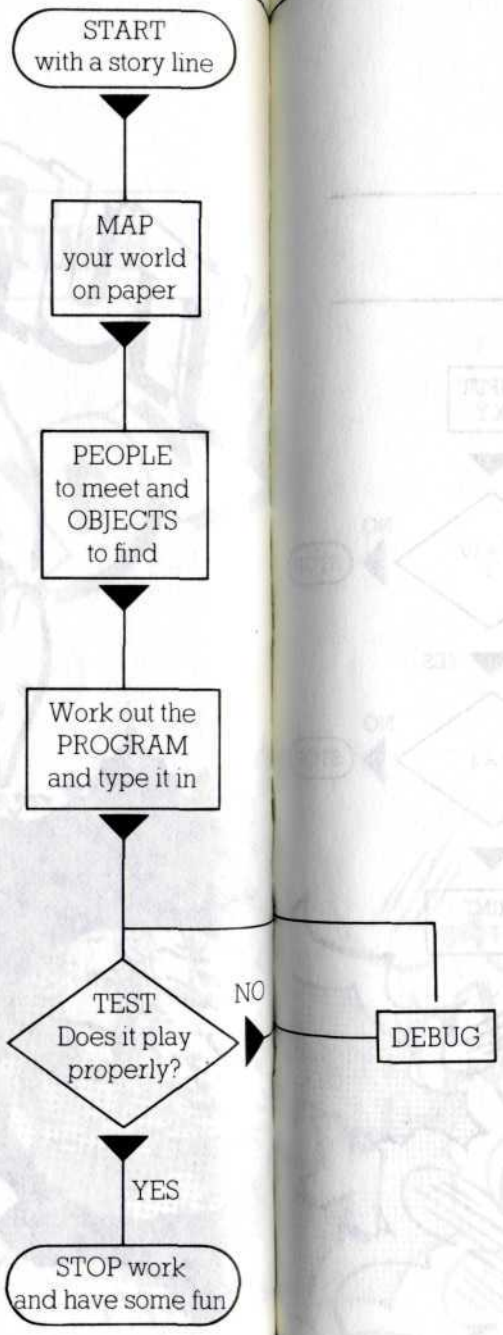
At the time of writing, probably the most successful adventure game around is 'The Hobbit' (Psion & Melbourne House), where the player enters Tolkien's wonderful world of dwarves and trolls, elves and goblins, magicians and spiders. Like proper adventure games, this is text-based, but some pretty pictures of different locations have been added – just to brighten things up.

## Writing an adventure game

Start with the storyline. What is the atmosphere that you want to create? Where will your adventure take place? What kind of quest is it? When you have got the feeling of what it's all about, and have sketched in the broad outlines, then it's time for the next stage.

Design your map. You are going to create a new world for your adventure, and that world needs lots of interesting places where unpredictable things can happen.

People your world with different characters for the player to meet. Some of them might be



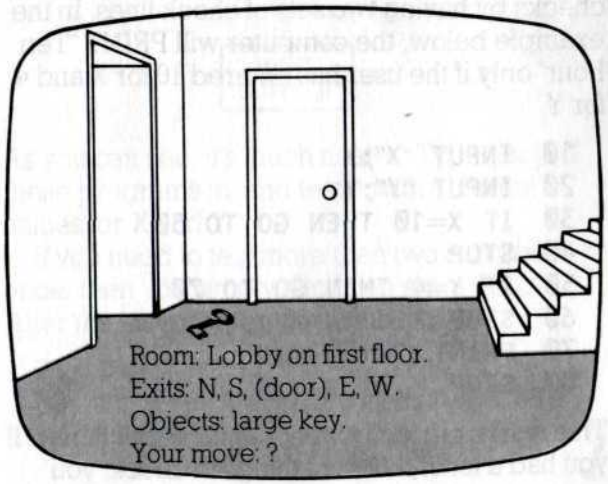


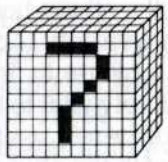

friendly, others hostile. You will also need to place a variety of objects in various locations, for the player to find. Some of these might be absolutely vital to the success of the adventure, others merely useful, and some totally useless. One of those objects should be the chest of treasure, magic ring, lost crown or whatever else you decide is the key object which must be found.

All of this needs to be planned out carefully on paper, well before you start to hit the keyboard. The more time and care you take at this stage, the quicker and less frustrating the programming will be.

The final stage of the writing process is testing the game. You will find, unless you are one of those rare perfect human beings, that there are things that you didn't foresee, and that your program needs all sorts of minor – or major – adjustments before the game goes as you wanted.

All in all, writing an adventure game is a long process, but an interesting one. We'll start to look at it in detail shortly. First, though, there are a few programming techniques that are vital in this type of game, so let's have a look at those.





# BASIC logic

You must have already come across the Conditional Jump:

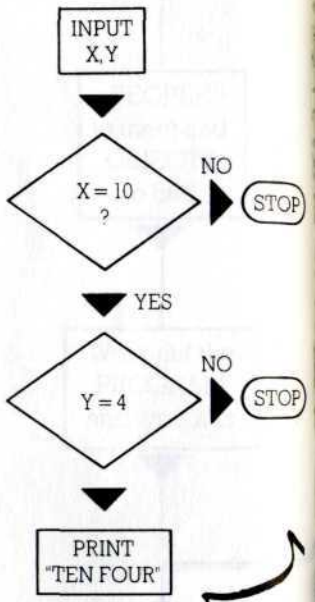
```
IF X=10 THEN GOTO......
```

As long as the single condition (X=10) is right, then the program jumps as instructed. When writing adventure programs you also need to be able to handle more complex conditional jumps. The computer has to check that both the verb and object are acceptable, and lead to some sensible action on its part. It needs to check instructions further: if the player has said he wants to light a torch, this can only be allowed if he has both matches and a torch.

You can cater for these kinds of double checks by having two sets of check lines. In the example below, the computer will PRINT "Ten Four" only if the user has entered 10 for X and 4 for Y.

```
10  INPUT "X";X
20  INPUT "Y";Y
30  IF X=10 THEN GO TO 50
40  STOP
50  IF Y=4 THEN GO TO 70
60  STOP
70  PRINT "Ten Four"
80  STOP
```

This works perfectly well, but it's very clumsy. If you had a lot of different things to check, you would have jumps all over the place.





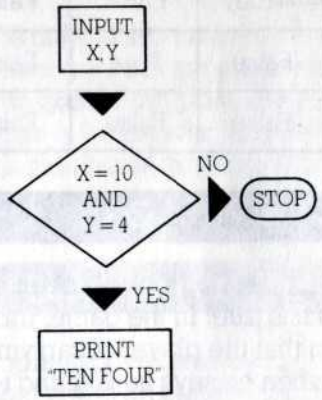


A neater and, in the end, simpler way to cope with two conditions is to use the LOGICAL OPERATORS. These are the words AND, OR and NOT.

## AND

This operator is used when you want the program to jump if both conditions are true. You can rewrite the "Ten Four" program like this:

```
10  INPUT "X";X
20  INPUT "Y";Y
30  IF X=10 AND Y=4 THEN PRINT
    "Ten Four"
40  STOP
```




As you can see, it's much neater. Type each of these programs in, and test it with different values for X and Y.

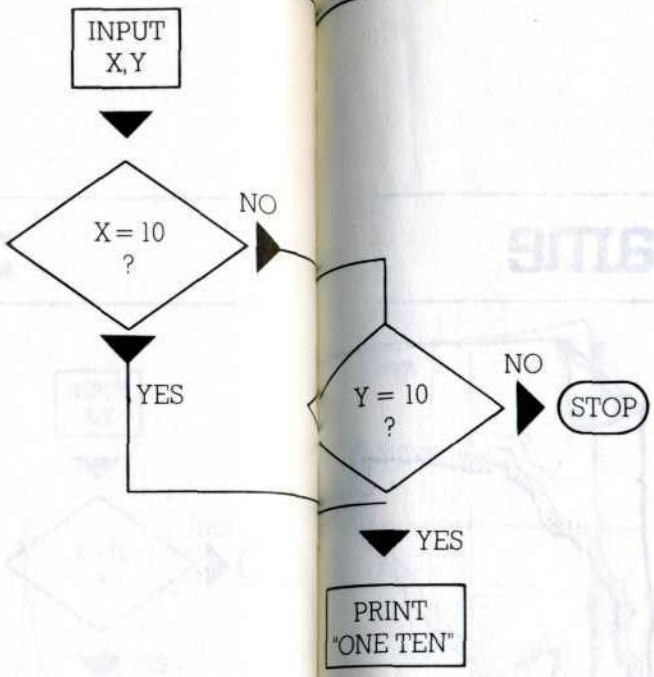
If you need to test more than two conditions at once, then you simply slip in an extra AND... Alter the last program to include these:

```
25  INPUT "Z";Z
30  IF X=9 AND Y=9 AND Z=9 THEN
    PRINT "Emergency!"
```

## Logical operators

Computer scientists like to show how logical operators work, by setting out all the possible combinations in a TRUTH TABLE. Here's the table for AND (when there are two conditions). You will see that both the X condition and the Y condition have to be true, before X AND Y is true. (The program only jumps if X AND Y is true.)

| AND | | |
|---|---|---|
| X | Y | X and Y |
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |




## OR

The OR operator looks to see if either of the conditions is true. In the game, this might mean checking that the player is carrying a gun OR an axe, when he says he is going to kill something. You can see it at work in this program.

```
10  INPUT "X";X
20  INPUT "Y";Y
30  IF X=10 OR Y= 10 THEN
    PRINT"One is a ten."
40  STOP
```

Type it in and try it with different values. What happens if both X and Y are 10? The OR operator leads to a jump if either X is true, OR Y is true OR both are true. You can see this in the truth table.

| OR | | |
|---|---|---|
| X | Y | X or Y |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

## String variables

All of the examples above have used number variables, but the logical operators work just as well with string variables. Try this:

```
10  INPUT "COLOUR 1";C$
20  INPUT "COLOUR 2";D$
30  IF C$="RED" AND D$="BLUE" THEN
    PRINT "MY FAVOURITE COLOURS."
40  STOP
```

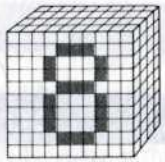
Now change line 30 and try it again:

```
30  IF C$= "RED" OR D$= "BLUE" THEN
    PRINT "I LIKE GREEN."
```

Note that line 30 only works properly if the same typefaces are used by both the player and the computer. It's no good if the player types in 'red' and the computer is looking for 'RED'. There are other problems like this with string variables, and we will return to them later.

## Equals or not

All of the examples above have used expressions containing an equals sign (X=10). The logical operators work with any of the comparison signs <,=,>,<>. Try typing in one of the short number programs above, and changing the check line so that the jump occurs when X is less than a certain limit, and when Y is not equal to a given number. There are lots of variations you can work out on these lines using AND, OR and the comparison signs.



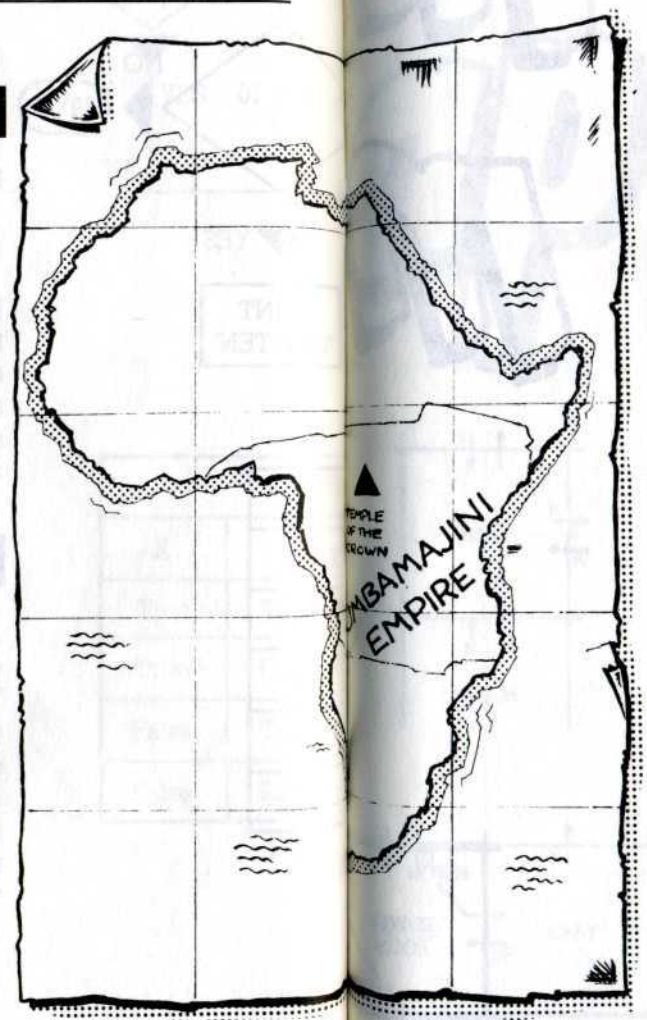
# Planning the game

## The story

What is the story behind your game? Where does the action take place, and what sort of things can happen. What is the point of the game? These are basic questions that you must answer before you can go any further.

Adventures based on Goblin-and-Dwarf fantasy land are traditional, but are perhaps a bit overdone. Haunted Houses and Science Fiction locations are quite common, but offer lots of scope for imagination. How about a search for the Bigfoot in the wild timberlands of America, or the Quest for the Holy Grail, set in Medieval Europe? You could even set up a detective-type adventure based on your own town, or another real area that you know quite well.

In JUNGLE, the story is this: deep in darkest equatorial Africa, there is a hidden temple, and somewhere in that temple is the ancient crown of the Emperor of Umbamajini. In ages long past, the Umbamajinian Empire stretched from coast to coast across that vast continent, and, according to the legend, whoever can find the Lost Crown will be able to claim that enormous empire, and all its wealth, as his own.






The adventurer sets out from Port Bata, on the coast of Equatorial Guinea, to explore inland regions beyond. Much of the land through which he will travel is thick jungle, but there are also mountains, swamps and desert. The adventurer has brought a number of useful things with him, but he does not have all that he will need if he is to survive and complete his mission. There are other essentials which he must find, somehow or other, as he goes on. For a start, he will need an axe if he is to chop his way through the impenetrable jungle.

The journey is fraught with danger. There are wild animals and cannibals out there in the jungle, and the explorer could easily come to an untimely, and sticky, end in a patch of swamp. There are also some valuable contacts to be made. Legend has it that the inner room of the temple is sealed with an enchanted door. Only he who knows the Word can open that door. The Guardian of the Word is an old hermit who lives somewhere in the mountains. Perhaps he could be bribed to part with his secret. . .

The object of the game is to find that Lost Crown. Only that way can the search come to an end. As an extra, and rather unfriendly, twist to the game, it is possible to destroy the crown accidentally, and not notice. The penalty for this is to be trapped forever in the game!

# The map

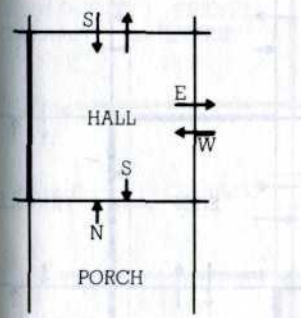
When you have got your storyline planned, and you are happy with it, then it is time to start on the map. This is the stage at which you will work out the details of the game.

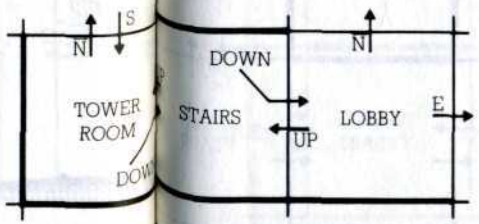
The size of your map – how many different areas or 'rooms' it has – depends upon how much memory you are prepared to allow for it. A lot of space will be needed for all the data that the game requires, and for the routines to handle the different events. In JUNGLE the map has 64 rooms, and the total program takes up approximately 8k, which is more than enough for anyone to key in. If you intend to make much use of the game, and you have a 48k Spectrum, then you should expand it later. The descriptions, and event-handling routines, have been much compressed to keep the program as short as possible.

The map should allow several good routes from the start to the goal and should include a number of dead ends, one-way systems and other obstacles. There should also be other places of interest to visit on the way. Some of these visits may be vital to the game – in JUNGLE, you have to go to the shop to get an axe before you can chop your way through the areas of impenetrable jungle.

You do not have to fill every space on the map. Rooms can be left empty and walled off from the player. This map is never visible on the screen, and sealed rooms could be a useful distraction for some players. The really serious adventurers try to draw up the map for themselves as they carefully explore each and every avenue. They could spend a long time trying to get into those closed off rooms!

Mark on the map the exits from each room. These do not have to be open, or even visible. You can have locked doors, darkened rooms where the exits can only seen when a torch is lit (or dense jungle where you have to chop down trees to see), or even hidden doors which must be searched for.




At some point you will have to convert these exits into the form of compass directions, as that is the way that the program will handle them. It may be useful to do it at this stage. If you want to give the impression that your mapped world has three dimensions – stairs in a house, a mountain climb, dungeons and towers in a castle, then you can label some of your exits Up and Down. You will then need a special routine to convert the Up/Down movements back into two dimensions. This is done in JUNGLE. You can even create a truly three-dimensional map, but more of that later.

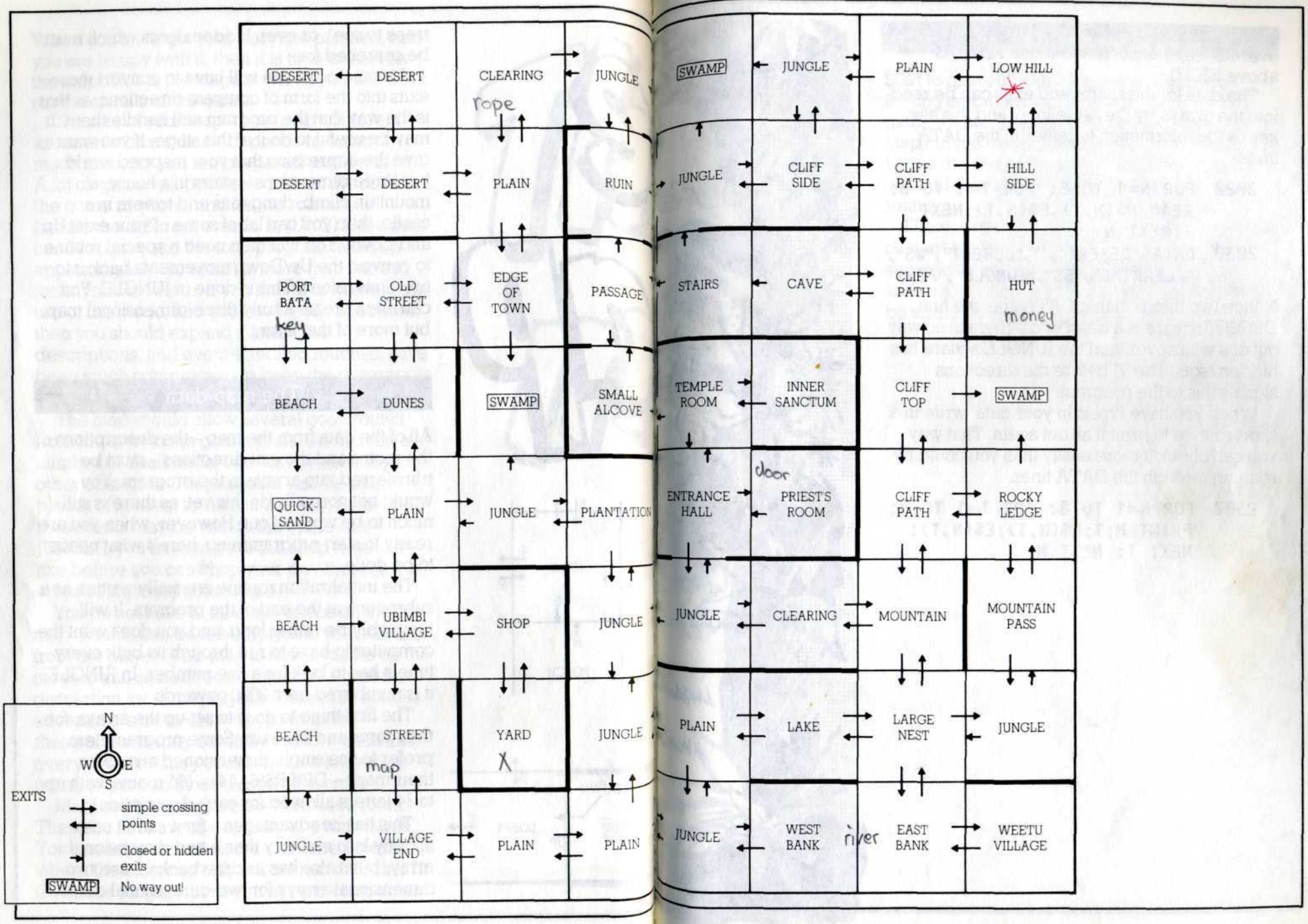
## The map data

All of the data from the map – the descriptions of the rooms and the exit directions – must be transferred into arrays in the program. You would not normally do this yet, as there is still much to be worked out. However, when you are ready to start programming, here's what needs to be done.

The initialization routine is usually written as a subroutine at the end of the program. It will inevitably be rather long, and you don't want the computer to have to run through its bulk every time it has to look for a line number. In JUNGLE, it is numbered from 2000 onwards.

The first thing to do is to set up the arrays, for the rooms and the exits. Some programmers prefer to use single dimensioned arrays for their maps – DIM R$(64,14) – (64 rooms, with up to 14 letters allowed for each description).

This has its advantages – for a start it uses slightly less memory than a two-dimensional array, but it also has its drawbacks. I use two-dimensional arrays for two-dimensional maps.


DESERT
DESERT
CLEARING
JUNGLE
rope
DESERT
DESERT
PLAIN
RUIN
PORT BATA
key
OLD STREET
EDGE OF TOWN
PASSAGE
BEACH
DUNES
SWAMP
SMALL ALCOVE
QUICK-SAND
PLAIN
JUNGLE
PLANTATION
BEACH
UBIMBI VILLAGE
SHOP
JUNGLE
BEACH
STREET
map
YARD
JUNGLE
JUNGLE
VILLAGE END
PLAIN
PLAIN
SWAMP
JUNGLE
PLAIN
LOW HILL
JUNGLE
CLIFF SIDE
CLIFF PATH
HILL SIDE
STAIRS
CAVE
CLIFF PATH
HUT
money
TEMPLE ROOM
INNER SANCTUM
CLIFF TOP
SWAMP
ENTRANCE HALL
door
PRIEST'S ROOM
CLIFF PATH
ROCKY LEDGE
JUNGLE
CLEARING
MOUNTAIN
MOUNTAIN PASS
PLAIN
LAKE
LARGE NEST
JUNGLE
JUNGLE
WEST BANK
river
EAST BANK
WEETU VILLAGE
N
W
E
S
EXITS
simple crossing points
closed or hidden exits
SWAMP
No way out!

It is very obvious that R$(4,5) is directly over R$(5,5), but not so clear that R$(29) is above R$(37).

The data for the rooms and exits can be read into the arrays by the same loop, and this also keeps the information together in the DATA lines.

```
2020  FOR N=1 TO 8: FOR T=1 TO 8:
      READ R$(N,T),E$(N,T):NEXT
      T:NEXT N
2030  DATA "DESERT","","DESERT","WS",
      "CLEARING","ES","JUNGLE","/EWS"
```

Notice two things in that DATA line: the first DESERT square is a dead end – there is no way out of it whatsover, and the JUNGLE square has hidden exits. The "/" before the directions signals this to the program.

When you have typed in your data, write in a short routine to print it all out again. That way you can check it more easily than you could by reading through the DATA lines.

```
2500  FOR N=1 TO 8: FOR T=1 TO 8:
      PRINT N;T;R$(N,T);E$(N,T):
      NEXT T: NEXT N
```



## Project

Start to plan out your own game. If you are feeling ambitious and intend to make it a big game – a hundred or more rooms – then only map out the main routes and areas at this stage. Leave the rest of the rooms empty, and sealed off, until you have got the essential routines running.

Draw up the map and work out the exits, then convert the information to a DATA list. You could write the subroutine to set up the arrays and read in the data now, and save it on tape, to be added to later.

# Objects

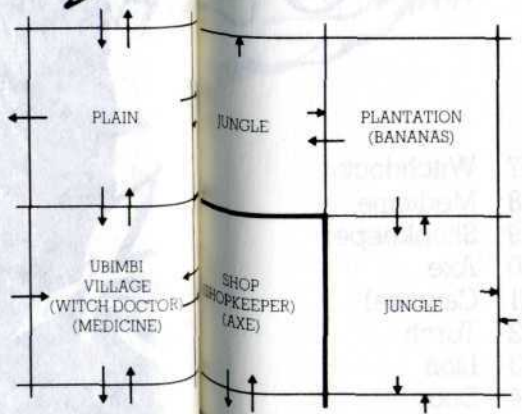



Your adventure games must have things for the players to find, and people, or 'things' for them to meet. If yours is a 'Tunnel of Gloom' type of game, where several different varieties of nasties are lying in wait for your luckless adventurer, then these could be scattered at random about the place, as in the maze games covered in the action games section.

If you want to create a more organized and thoughtful type of adventure, then you will need to spend more time working out where you are going to place what, and for what reason. There should be some objects which are vital to the success of the game – the key to the door, the matches and torch to light the darkened rooms – and others which will give your players something to think about. Their locations are also very important, and cannot be left to random chance. You could not have the matches hidden at the end of the darkened tunnel, as the player could never reach them. Lastly, the objects can be characters in the story.

You can see examples of all of this in these six squares from the JUNGLE game.

When the player enters the PLANTATION, he will find BANANAS. He can take these, and eat them if he likes. This might be tasty, but won't do him a lot of good. He really needs to keep those bananas with him in case he meets a GORILLA, and then they can be put to good use.






The SHOP has a SHOPKEEPER visible in it. If the player offers money or trade goods to the shopkeeper, he will be given an AXE. The player must have the axe before he can enter any JUNGLE square.

Next door to the SHOP is UBIMBI VILLAGE, where the player will meet a WITCHDOCTOR. He plays no meaningful part in the game, and the MEDICINE, which he will give you in exchange for any trade goods, is quite useless. The player is not to know this. You can write an interesting variation to the game by adding an INVENTORY routine. This takes note of all the things that the player is carrying, and can be used to limit the number of items which can be carried. This forces the player to make guesses about what is, and what is not useful.

ALL of the JUNGLE squares contain JUNGLE, which is treated as an object by the program, but handled by a special routine. It is not necessary, therefore, for JUNGLE to be marked in every square.

The word 'Objects' is used here for good grammatical reasons. When the player gives his move to the computer, it is in two words (generally) – Verb followed by Object. The program will need a routine to check the move to make sure that it is valid, and part of this check will involve looking at the object word and comparing it with the ones it knows. To enable it to do this, the objects need to be written into an array. The locations of the objects also need to be stored in an array, and if, as happens in JUNGLE, some objects are visible and others are hidden, you need some means of indicating this.

If you look at the listing, you will find that line 2010 sets up a number of arrays, including these – O$(),P() and F(). O$() is for the OBJECTS, P() is for their positions, given as coordinates, and F() is a set of FLAGS. These are mainly used to

indicate the state of the objects – are they visible, hidden, or being carried by the player?

The main list of objects is given here. There is some order to the way that they are organized.

The first object is JUNGLE, which appears to be at 0,0 but in fact is in every appropriate square. It has to be included in the object list because one of the valid moves (and the one which is probably used more than any other) is 'CHOP JUNGLE'.

Objects 2 to 8 are all things which can be found lying around in various squares, and which can be carried off by the explorer.

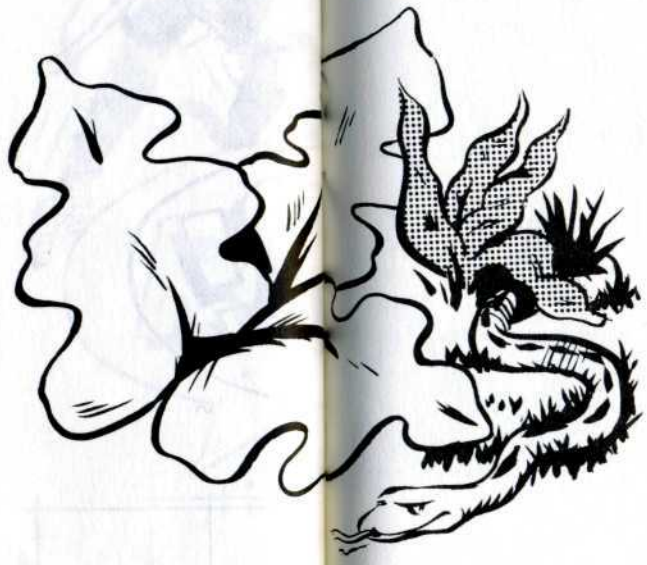
The RIVER is a bit like JUNGLE. A twist in the program allows it to appear in (or between) two squares, 8,6 and 8,7 and while it can't be carried off, 'CROSS RIVER' is a good move.

The set of objects from 11 to 22 are arranged in pairs. The first of each pair – GORILLA, HERMIT, SNAKE, WITCHDOCTOR, SHOP-KEEPER, and CANNIBAL – is visible in the given square. If the player does the right thing to, or with, this character, then he is rewarded by getting the second object of the pair. To save memory space, most of these follow the same routine. The player gives a present, and receives one in exchange. There are exceptions to this. The gorilla only likes bananas, and the snake has to be killed, in the right way, to uncover the crown. Also, you can eat the cannibal and still find the torch.

**Object list**

| | | | | | |
|---|---|---|---|---|---|
| 1 | Jungle | 9 | River | 17 | Witchdoctor |
| 2 | Key | 10 | Bananas | 18 | Medicine |
| 3 | Gun | 11 | Gorilla | 19 | Shopkeeper |
| 4 | Beads | 12 | Stick | 20 | Axe |
| 5 | Cloth | 13 | Hermit | 21 | Cannibal |
| 6 | Money | 14 | Word | 22 | Torch |
| 7 | Boat | 15 | Snake | 23 | Lion |
| 8 | Matches | 16 | Crown | 24 | Door |





# Verbs

What sort of actions are you going to allow in your game? The adventurer must be able to go from place to place, to pick things up and to use them in various ways. He might also be able to speak, search for hidden objects, light torches, unlock doors, kill monsters and other enemies, eat and drink, and cast magic spells. The limit to the number of possibilities in your game is the amount of memory space that you are prepared to leave for the routines which manage the actions. In JUNGLE these take very nearly half of the total program. The verb list itself is very short:

LOOK, GO, TAKE, GIVE, SAY, OPEN, EAT, LIGHT, CHOP, CROSS, KILL.

Each of these requires its own routine, of anything from three to ten program lines. Some verbs are very simple to handle. 'LOOK' will normally just send the program back to the point where it prints a description of the room that the adventurer is in at that time. Sometimes it will also reveal objects that are normally hidden, but to handle this, it is only necessary to check the coordinates of the current location against those of the squares with hidden objects.

Other verbs are more complex. When the program meets a GO instruction, it must check that the verb is followed by a valid direction word (N,E,S,W,U,D), that the exit is visible and that it is possible to GO in the given direction. If all is well, then the instruction must be

converted into a change of coordinates, and, if not, then the player needs a message to tell him why not.

When you are working out your verb list, think about the acceptable Verb-Object combinations at the same time.

| VERB | OBJECT | RESULT |
|---|---|---|
| LOOK | none needed | Show location, + hidden objects if in right squares |
| GO | N,E,S,W,U,D | Move if exit visible, present and open |
| TAKE | Portable objects | Flag to show being carried |
| GIVE | Any carried | Receive object, if in right square |
| SAY | the WORD | Open enchanted door, if WORD carried |
| OPEN | DOOR | Unlock door, if KEY carried |
| EAT | BANANAS<br>CANNIBAL | 'Yum yum' message<br>Reveal hidden TORCH |
| LIGHT | TORCH | Shows exits, if TORCH and MATCHES carried |
| CHOP | JUNGLE | show exits from jungle square |
| CROSS | RIVER | move between 8,6 and 8,7 |
| KILL (WITH) | GUN<br>AXE<br>STICK | Depends on place and victim |

You will notice that the 'Objects' listed next to GO are the directions, and these need to be added onto the Object List. In JUNGLE, the O$() array is dimensioned for thirty objects (twenty-four things and characters, and six directions). If

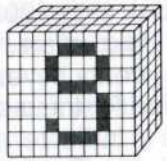


you look at line 2010, you will also see that the P() – Position array, and F(), the Flags, both have twenty-seven stores. Twenty-four flags are needed for the major objects, and a further three to indicate visibility, and the status of the two doors. The P() array is really slightly larger than necessary, but was made that size to allow a simple READing loop.

```
2110 FOR N=1 TO 30: READ O$(N):
     IF N<28 THEN READ P(N,1),
     P(N,2),F(N)
```

KILL is the most complex of the verbs used in the JUNGLE game. It must be used in the form 'KILL WITH GUN/AXE/STICK', and can only be used where a second character or animal is present. In the version given here, the KILL instruction always works where the victim is harmless (GORILLA,HERMIT, WITCHDOCTOR and SHOPKEEPER), although none of these should be killed, as each has something to give to the adventurer. When the player tries to kill the LION,SNAKE or CANNIBAL, it becomes a bit more risky. There is a chance that he will miss, and then he gets eaten. Finally, as part of this routine, the SNAKE has to be killed with the right thing if the Lost Crown is to be revealed.

Think through the possible complexities of the Verbs before you commit yourself to them. It would probably be best, in the early stages of the game, to keep to an absolute minimum. You can always expand the game later when the essential routines are working properly.



# The key routines

## The program core

Adventure games will generally follow the same pattern, and use the same, or very similar, routines at their core. They are made individual by the data that goes in at the initialization stage, and by their action subroutines.

## The print routine

This starts by telling the player where he is and what he can see. How you present this is important. Straightforward text printing, as in JUNGLE, can get boring, unless you have a gift for words and can write some stunning descriptions of the rooms. Pictures can be added here if you are prepared to take the trouble. They do, of course, take up memory, and a separate picture for each location would eat up space alarmingly. Limiting yourself to a different picture for each type of location will help – in JUNGLE there are 40 types of rooms in those 64 squares. An alternative is to give pictures only for certain locations – the more interesting ones, or those where an object is to be found.

Divide the screen into picture area and text area and keep all your printing within the limits.





This can create a minor problem. If the player stays in one location for any length of time, while he tries out different instructions (perhaps looking for a way to escape), then there is going to be a build up of text. You can cope with this in several ways.

**1** Restrict the amount of text. Only display the location, and other basic information, and not the previous moves. This is a simple way to keep the screen tidy, but some players like to be able to see at least some of their most recent moves.

**2** Store the last few moves and display them separately. The display can be updated by a simple loop. Here the M$() array stores the last five moves.

```
4000  FOR N=1 TO 5
4010  LET M$(N)=M$(N+1): NEXT
4020  RETURN
```

The latest move is then stored in M$(6).

NOTE A very friendly option to offer your players is a 'Move Review'. If the space is available, you could store every move in an array, and display the move number at each go. The player could then choose to see any particular move or set of moves, or every move that he has made. This could be especially useful at the end of the game, where he tries to find out where he went wrong!

**3** Create a limited scroll. You can get the computer to shuffle lines of print up the screen, while the rest stays fixed, by getting it to read

the characters and rewrite them on another line. This is the kind of routine that you would need:

```
10  CLS
20  FOR Z=1 TO 5:PRINT Z:NEXT Z
30  FOR L=1 TO 5:FOR C=0 TO 10:
    LET X$=SCREEN$(L,C)
40  PRINT AT L-1,C;X$:NEXT C:
    NEXT L
50  PRINT AT 4,0;Z: LET Z=Z+1
60  GOTO 30
```

Line 30 starts reading the screen from the second line down, storing the character in X$. This is then displayed directly above by line 40. Line 50 then prints the next Z number in the space created below to keep the demonstration going. Only numbers are used here but this technique can be used for as much text as you want, anywhere on the screen.

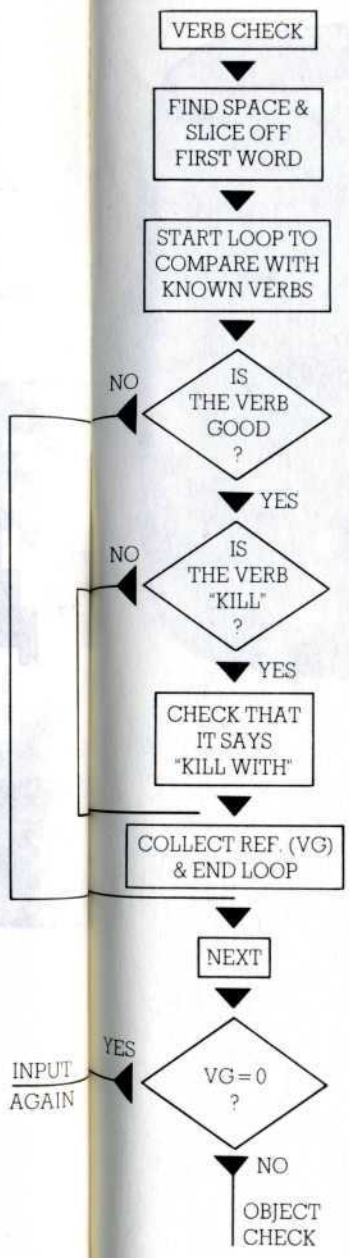
## Syntax checks

Checking that the player has used words that make sense to the computer is largely a matter of string slicing and logic. The routine that handles this in JUNGLE runs from line 150 to 300.

It starts by separating the first word from the rest of the string. This is done by finding the space between the words and slicing off the portion to the left of the space.

```
150  ...FOR N=1 TO LEN A$:
     IF A$(N)<> " " THEN GOTO 170
160  LET D$=A$(TO N-1):LET Q=N:
     LET N=LEN A$
170  NEXT N
```






At the end of the routine, D$ holds the first word and Q stores the position of the space. This will be needed later.

The next stage is to run the verb array through a loop, comparing each in turn with D$. When the word is found, its reference number is stored in VG (Verb Good). If at the end of the loop VG = 0 (to which it was set at the start) then the computer knows that no match has been found.

If you are using verbs which do not need objects – like LOOK, or RUN – then now is the time to check for them. Keep these single word instructions at one end of the array so that handling them is easy. . . .

```
IF VG=1.....
```
(if the word is LOOK)

Next make sure that there is another word there. . . .

```
IF Q=LEN A$ THEN....
```

You must not try to slice a word that isn't there, as the program will crash, and the next stage is to slice off the second half of the input and transfer it to another store.

```
LET W$=A$(Q+1 TO LEN A$)
```

W$ is then compared with the words in the object list, and its reference number held in KW (Known Word).

The final section of the main program sends the computer off to the action subroutines, and then checks for any possible endings before looping back to the PRINT lines.

In JUNGLE, there are several ways in which the program can be ended. The adventurer can be lost in the desert, or walk into a patch of jungle without an axe, disappear in quicksand or swamp, or be eaten. Lines 340 to 370 deal with these possibilities. There is even a chance that your player might actually reach the goal

and find the crown. This is checked in one of the subroutines, and the program ends at that point. This is not good programming practice, and is not to be encouraged. Ideally you should always return from subroutines, and end your program from the main routine. You must do this if you are going to offer the player another game from within the program. Leaping out of subroutines without using RETURN leaves the return line number on the Gosub Stack, and uses up valuable memory space. You can see the limit of the Gosub Stack by running this program:

```
10  LET N=0
20  PRINT N,
30  LET N=N+1: GOSUB 100
100 GOTO 20
```

How many return addresses can it store before overflowing?

## Project

Start to key in the JUNGLE program. You will need all of the lines from 2000 onwards to set up the arrays, and the main routine down to 380. You must also type in RETURN lines at 1150, 1200, 1300, 1350, 1400, 1450, 1500, 1550, 1600, 1650, 1700 and 1750. This will give you enough to test the main program. It's as well to get that right before you tackle the action subroutines.

Use the main routine given here in your own game. The only things that you will have to change are the numbers in the loops, so that they agree with the sizes of your arrays.



The variables

| | |
|---|---|
| R$() | Room ARRAY |
| PL, PC | Player's Line and Column co-ordinates |
| GE | Giver or Enemy – used to store ref. number when a character or animal is met. |
| CS | Can See something? (0 = No. 1 = Yes) |
| VE | Visible Exits? (0 = No. 1 = Yes) |
| E$() | EXIT ARRAY |
| F() | Flag ARRAY. F(25) is general visiblity flag. F(25) = 0 means you can't see anything. |
| Q | Stores position of space in instructions. |
| D$ | Do something – the Verb. |
| VG | Verb Good. VG = 0 means word not known. |
| W$ | Word – do What. The object. |
| KW | Known Word. KW = 0 – unknown object. |
| EU | Eat You. EU = 1 means that something or someone has eaten the adventurer. |

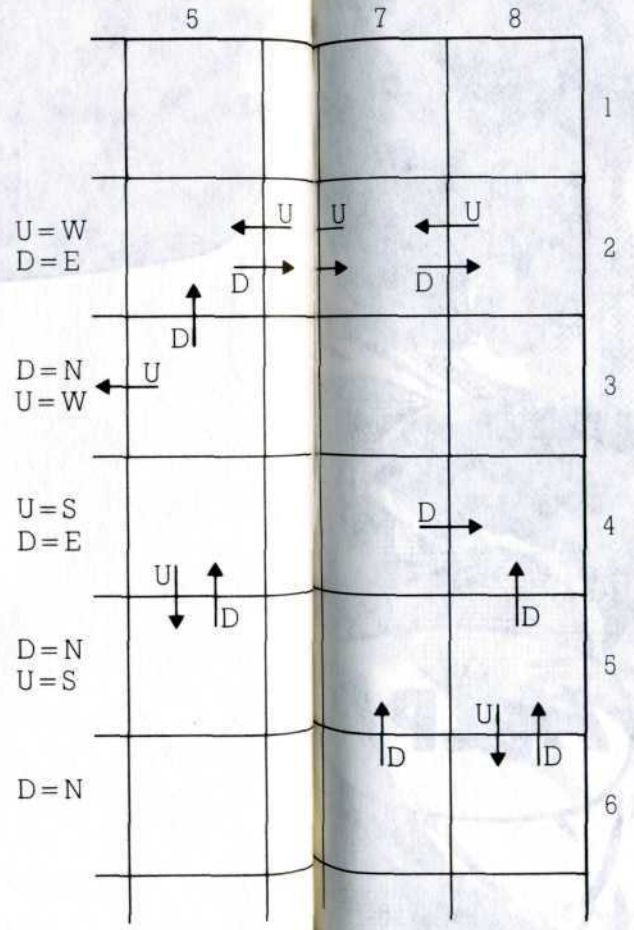
# Action subroutines

Action subroutines are all about Condition Testing. Has the player used a meaningful combination of words? Is he in the right place to perform a particular action? Has he all the equipment he needs to be able to do it?

Look at the routines in the JUNGLE listing to see how they are tackled. They are not easy program lines to read, as they contain so many reference numbers and variables. Take, for example, the GO subroutine – your adventure games will need one very similar to this.

```
1200  IF KW<25 THEN PRINT "I CAN'T
      GO "; W$: RETURN
1210  LET G$=O$(KW,1):FOR N=1 TO
      4: IF E$(PL,PC,N)="/" AND F$
      (25) ="0" THEN PRINT "I CAN'T
      SEE TO GO ANYWHERE,": RETURN
1220  IF G$<>E$(PL,PC,N) THEN GO
      TO 1290
1230  IF PC=5 AND (PL=4 AND
      F(26)=0) OR (PL=6 AND
      F(27)=0) THEN PRINT "THERE
      IS A CLOSED DOOR.": RETURN
1240  LET PC=PC-(G$="W")+(G$="E")
1250  LET PL=PL-(G$="N")+(G$="S")
1260  IF G$="U" THEN LET
      PC=PC-(PL<4): PL=PL+(PL>3)
1270  IF G$="D" THEN LET
      PC=PC+(PL=2 OR PL=4): LET
      PL=PL-(PL=3 OR PL>4)
1280  LET N=4: NEXT N: RETURN
1290  NEXT N:PRINT"I CAN'T GO THAT
      WAY.": RETURN
```

The first thing to check here is that the player has given a proper direction. All of these have reference numbers of 25 or more. (Line 1200.) If you wanted to include witty responses to any illegal commands they would have to be fitted in before this general cut-out line. For example, you might not want to let a GO BANANAS command pass without comment.




The first part of line 1210 is not strictly necessary. It transfers the array variable to a simple variable for ease of handling.

As so many of the rooms in this game have hidden exits, it makes sense to check that the player can see to go, before you do anything else. "/" indicates that the exit is hidden, and if F(25)=0 then either the jungle hasn't been chopped down, or the torch has not been lit.

The last stage before allowing a move is to check that the player is not trying to walk through locked doors. These are at 4,5 and 6,5, and their status is flagged by F(26) and F(27).

Normal N,E,S,W movements are a simple matter of using the logical operators to add to or take from the line or column variables. There are then two extra lines to convert 'U' and 'D' into two-dimensional moves. The way it works in JUNGLE is that Up means West for any room above line 3, and South for any room below. Likewise, Down means East on lines 2 and 4, and North on line 3 and below line 4. If you are using Up and Down, then try and arrange it to keep the conversion simple.

Notice that there are two RETURN lines. If, and when, the program finds a valid move, it will come to line 1280, close the loop and return. If it runs right through the loop without finding a match between your chosen direction and the possible exits from that room, then it returns after printing the 'No go' message.

## Project

Start writing the subroutines to handle the actions in your own game.

# 10 Taking it further

JUNGLE is a very basic example of adventure games. There are a great many ways that it could be improved. You might like to do this instead of, or as well as, writing your own game from scratch.

## Help

Struggling players will very much appreciate a HELP option. It can take several forms, but whatever kind of help you give, it should be accessed in the same way. Include HELP in your Verb list, and write in a subroutine to handle it.

## The help page

Here the normal location text display is replaced by a complete page of general help and advice. As an absolute minimum this should include a list of the acceptable Verbs and Objects, and a note on how to enter commands. You can go further and give a list of all the acceptable combinations of words (if there is space on the screen).

If you intend to give more specific help, then it is better done a different way.




## Helpful hints

In this form, the program gives the player advice on how to cope with the particular situation he finds himself in. The hint will appear as a single line in the normal text area. It might be a clue, or it might be the actual solution to a problem. If you have the memory space to spare, you could set up an array to hold helpful hints for every room, and read these into the array at the same time as the program is reading the room descriptions and exits.

A more economical form of help-giving is to have a short list of general hints.

1 TRY GOING NORTH
2 TRY GOING SOUTH
...etc
7 HAVE A GOOD LOOK ROUND
8 USE THE STICK
9 DON'T KILL HIM
10 THIS IS A USEFUL OBJECT

You can then create an array to hold the reference numbers of the appropriate hint for each room.

## Wizards, Warriors & Dwarves

Some adventure games are played 'Dungeons and Dragons' style. In this game each player has a different character, with a different balance of abilities. Warriors are strong, but not very bright, and can use only the simplest spells. Wizards have much magic at their disposal, Burglars are sneaky, Dwarves are tough and can keep a few spells up their sleeves. The game is usually played by groups of people, acting as a single party as they explore their way around the dungeons, and they are then able to put their different abilities to work in different situations. A feature of Dungeons and Dragons is that the dungeons are inhabited by a weird variety of monsters, some of whom can be best defeated by Warriors, others by Wizards, and others by Dwarves.

To develop a game along these lines you are going to need a whole set of new routines. At the simplest level, the game could be played by one player with his character determined by chance at the start. This can be done in either of two ways. The character type can be selected at random, with his balance of abilities fixed by the character, or the level of his different abilities can be decided by a set of RND lines, and a character given according to the balance. Of the two, the first method is by far the easiest to organize.

The levels of his abilities are held in variables, and brought into play during encounters. When player meets monster, and decides to fight, he does it by rolling the dice. The score needed to defeat the beast will depend upon the nature of the monster, and the level of the player's abilities. This combination of chance and known quantities enable the player to plan his moves more wisely. In the example here, any player with a strength of 9 or more knows that the odds are in his favour. He is most likely to kill the wolf (with a score of 7 or more) on the first blow, but even a very low score will inflict 50% wounds, so that a second stroke would finish it off.



The player's ability levels change during the course of the game. He gains in magic and strength as he overcomes challenges, and he is weakened by combat. The object of the game is to survive, and grow, as well as to find treasures.

As you can imagine, the action subroutines to cope with these encounters can become very complex. When working out this type of game, keep it as simple as possible at first. Develop the routines you need to handle the encounter with one type of monster – the ability variables, the Kill/Wound array, the 'dice-roller' – and then expand it, one aspect at a time.

| | Strength | Intelligence | Stealth | Magic |
|---|---|---|---|---|
| Wizard | 24 | 8 | 4 | 10 |
| Warrior | 10 | 6 | 2 | 2 |
| Dwarf | 6 | 6 | 2 | 4 |
| Burglar | 4 | 4 | 8 | 0 |

Wolf encounter

| Strength | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Kill score (>) | 12 | 12 | 12 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| 50% wound | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 1 |

Other characters can be introduced to the game in different ways. There can be predetermined companions, as in the famous HOBBIT program, and these can take a more or less active part in the game. Their abilities can be added to the player's, when they are around, and they can be made to appear and disappear at intervals. There are several ways in which you could handle this. Their presence can be controlled by random lines in the main loop:

```
IF RND>.9 AND COMPANION=0 THEN
LET COMPANION =1
```

You can mark certain squares as crossroads where the companion will branch off, if he was with the player, and join him if he wasn't.

Their stay can be timed. Reset the timer at the beginning of the game, and have the companion wandering off after a given time.

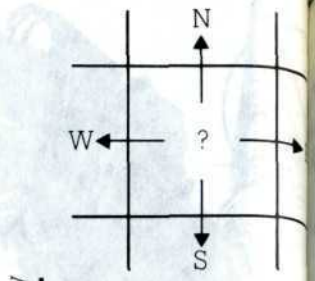
```
POKE 23672,0: POKE 23673,0
```

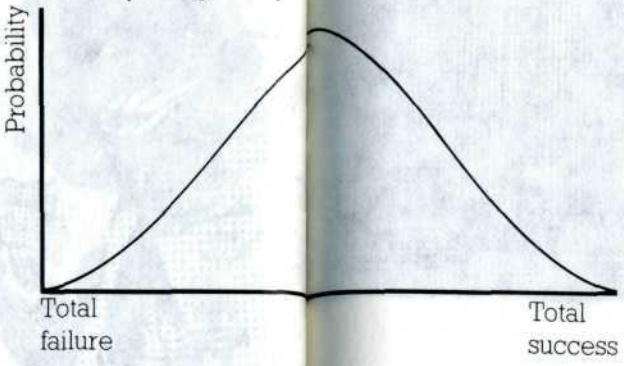
This would check for a lapse of 15 minutes:

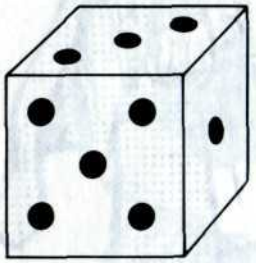
```
IF PEEK 23673>180 THEN....
```

These same techniques can be used to create moving monsters in your game. They don't actually have to move from square to square, simply to appear – preferably when least expected.

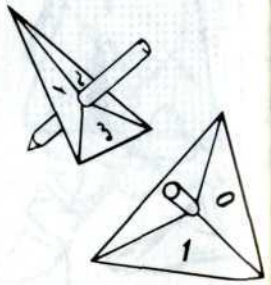
The companion characters can be brought under the control of the player, or of a group of players. At the INPUT stage, the computer would need to be told which character was to be affected by the following command, and the variables that hold information about the player must be made into arrays to cope with the extra characters. Apart from that, the rest of the game runs the same way that it does for a single character.



| Number | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Chance | 1/6 | 1/6 | 1/6 | 1/6 | 1/6 | 1/6 |



Spinner 1 / Totals

| | | | |
|---|---|---|---|
| 3 | 3 | 4 | 5 |
| 2 | 2 | 3 | 4 |
| 1 | 1 | 2 | 3 |
| | 0 | 1 | 2 |

Spinner 2

| Number | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Chance | 1/9 | 2/9 | 3/9 | 2/9 | 1/9 |



## Random factors

The simple random number is not always enough. It's fine where you want something to happen once every so many times:

```
IF RND>.8 THEN....
```

Where you have a distinct set of possible outcomes you can use lines like these:

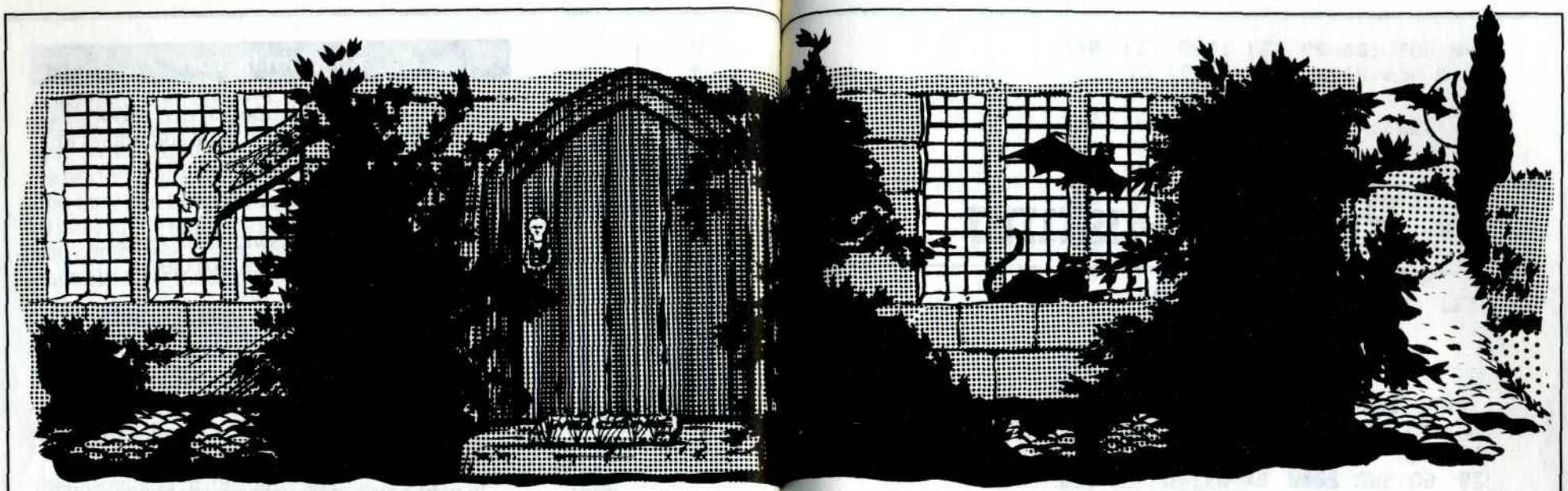
```
LET X=RND
IF X <=.25 THEN.. (go North)
IF X >.25 AND X <=.5
  THEN .. (go East)
IF X >.5 AND X <=.75
  THEN.. (go South)
IF X >.75 THEN.. (go West)
```

In your encounters with monsters you might want a more flexible, and realistic, outcome. Producing a random number on the computer is like rolling a dice. Every number is as likely as the next. What you want is for the middling numbers to come up more often than the extremes. Something in between is always more likely than total failure or total success.

You can create the 'curve' effect in two ways. One way is to set the limits for the random numbers so that the central section is wider:

```
IF X >.3 AND X <.7 THEN..
```
(50% wounds)

The alternative is to combine two random numbers. The most likely number you would get by rolling two dice is 7. This can turn up in six different ways. The extreme numbers, 2 and 12, will only turn up about once in every thirty-six rolls. For a percentage result – perhaps the percentage of wounds in an encounter – use two random numbers between 0 and 50. Over half of the totals will be between 35% and 65%.



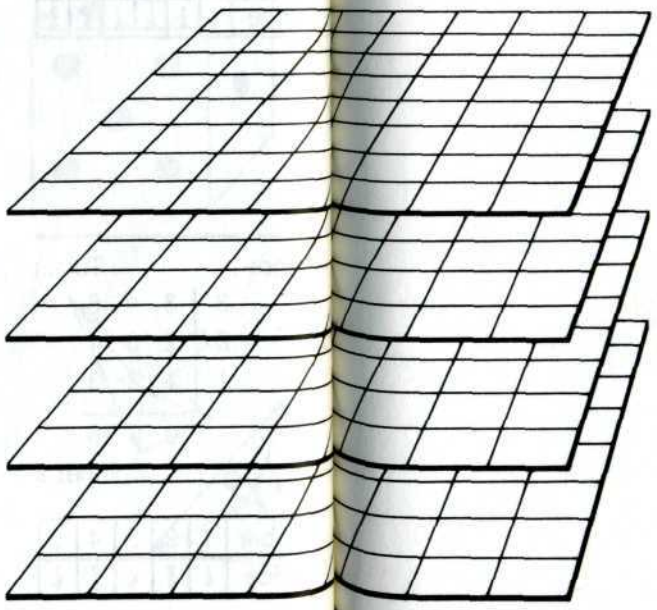
## 3-D maps

Running an adventure game in three dimensions is no more difficult than running one in two dimensions. The only extra programming involved is a pair of lines to handle the change of 'floor'. You will need to take a little extra care when setting up your maps, to make sure that the different floors agree with each other. The stairs going up from one level must meet the ones that come down from above.

The room array will need an extra dimension, and you will need to adjust the initialization stage so that all your data is read in, and in the right order.

If your game takes place within a building, then you will probably waste little space within the array. Each room will be used. On the other hand, if the game is set in a wider landscape, with mountains and castles rising out of flat land, then there could be quite a lot of the array left blank – unless your hero can fly! It is always worthwhile to stop at an early stage of your planning, and try to work out how much memory your game will need. As a very rough guide, in the JUNGLE program, about 1½k was used for the main program loop, 3k for the action subroutines and 2k for the initialisation. A further 2k was needed to create the arrays.

R$(F,L,C) Floors, Lines, Columns



The memory cost of an array can be calculated accurately. A string array uses 1 byte for each element, plus a few bytes for labelling. R$(10,10,10) will take approximately 1k, and will allow you up to ten letters per room for descriptions. If you created space for descriptions of fifty letters (about seven words) then a hundred room map would still only need 5k. The DATA lines to fill the array would probably need another 5k. You could cut this cost by SAVEing the array DATA separately and LOADing it in at the start of the game.

# Program listing




```
10  REM "JUNGLE"
20  GO SUB 2000
30  PRINT " LOCATION ";R$(PL,PC)
40  PRINT " YOU CAN SEE... "
50  LET GE=0: LET CS=0: FOR N=1
    TO 24: IF P(N,1)<>PL OR
    P(N,2)<>PC THEN GO TO 70
60  IF F(N)=1 THEN PRINT O$(N):
    LET CS=1: IF N>10 AND N/2
    <>INT (N/2) THEN LET GE=N
70  NEXT N: IF CS=0 THEN PRINT
    " NOTHING."
80  PRINT " VISIBLE EXITS "
90  LET VE=0: FOR N=1 TO 4: IF
    (E$(PL,PC,N)="/" AND F(25)=0)
    OR E$(PL,PC,N)=" "
    THEN LET N=4: GO TO 110
100 PRINT (E$(PL,PC,N) AND
    E$(PL,PC,N)<>"/"): LET VE=1
110 NEXT N: IF VE=0 THEN PRINT
    "NONE"
120 INPUT "WELL....?";A$
130 IF A$="STOP" THEN GO TO 1070
140 IF LEN A$<6 THEN LET A$=
    A$+" "
```



```
150 LET Q=1: LET D$=A$: FOR N=1
    TO LEN A$: IF A$(N)<>" "
    THEN GO TO 170
160 LET D$=A$( TO N-1): LET
    Q=N: LET N=LEN A$
170 NEXT N: GO SUB 3000
180 LET VG=0: FOR N=1 TO 11: IF
    D$<>V$(N) THEN GO TO 220
190 IF D$<>"KILL" THEN GO TO 210
200 IF A$(Q+1 TO Q+4)="WITH"
    THEN LET D$=D$+" WITH": LET
    Q=Q+5
210 LET VG=N: LET N=18
220 NEXT N
230 IF VG=0 THEN PRINT "I DON'T
    KNOW HOW TO ";D$: GO TO 120
240 IF VG=1 THEN GO TO 310
250 IF Q=LEN A$ THEN PRINT ' "TELL
    ME AGAIN. ";D$;" WHAT ? ":
    GO TO 120
260 LET W$=A$(Q+1 TO LEN A$):
    GO SUB 3020
270 LET KW=0: FOR N=1 TO 30: IF
    W$<>O$(N) THEN GO TO 290
280 LET KW=N: LET N=31
290 NEXT N
300 IF KW=0 THEN PRINT "'I DON'T
    KNOW THE WORD ";W$: GO TO
    120
310 GO SUB 1100+50*VG+(50 AND
    VG>2)
320 IF F(25)=1 THEN LET F(25)=2:
    GO TO 340
330 IF F(25)=2 THEN LET F(25)=0
340 IF R$(PL,PC, TO 9)
    ="QUICKSAND" OR R$(PL,PC, TO 5)
    ="SWAMP" THEN GO TO 1000
350 IF PL=1 AND PC=1 THEN GO TO
    1040
360 IF F(20)<>3 AND R$(PL,PC, TO 6)
    ="JUNGLE" THEN GO TO 1040
```

```
370 IF EU THEN GO TO 1020
380 GO TO 30
1000 PRINT R$(PL,PC)': GOTO 1060
1020 PRINT "YOU HAVE BEEN EATEN .":
     GO TO 1060
1040 PRINT "LOST.STARVED."
1060 PRINT "HARD LUCK.YOU'RE DEAD."
1070 PRINT "USE ""RUN"" TO
     RESTART.": STOP
1150 IF PL=2 AND PC=8 THEN LET
     E$(2,8)="UNS": RETURN
1160 IF PL=3 AND PC=6 THEN LET
     E$(3,6)="EW": RETURN
1170 IF PL=2 AND PC=4 THEN LET
     F(2)=1: RETURN
1180 RETURN
1200 IF KW<25 THEN PRINT "I CAN'T
     GO ";W$: RETURN
1210 LET G$=O$(KW,1): FOR N=1 TO
     4: IF E$(PL,PC,N)="/" AND
     F(25)=0 THEN PRINT "I CAN'T
     SEE TO GO ANYWHERE.": RETURN
1220 IF G$<>E$(PL,PC,N) THEN
     GO TO 1290
1230 IF PC=5 AND ((PL=4 AND
     F(26)=0) OR (PL=6 AND
     F(27)=0)) THEN PRINT "THERE
     IS A CLOSED DOOR.": RETURN
1240 LET PC=PC-(G$="W")+(G$="E")
1250 LET PL=PL-(G$="N")+(G$="S")
1260 IF G$="U" THEN LET
     PC=PC-(PL<4): LET
     PL=PL+(PL>3)
1270 IF G$="D" THEN LET
     PC=PC+(PL=2 OR PL=4): LET
     PL=PL-(PL=3 OR PL>4)
1280 LET N=4: NEXT N: RETURN
1290 NEXT N: PRINT "I CAN'T GO
     THAT WAY.": RETURN
```



```
1300 IF PL<>P(KW,1) OR PC<>P(KW,2)
     OR F(KW)=0 THEN PRINT " TAKE
     WHAT?": RETURN
1310 IF KW>22 THEN PRINT "THAT'S
     NONSENSE!": RETURN
1320 IF KW=1 OR (KW>10 AND KW/2
     <>INT (KW/2)) THEN PRINT
     "YOU CAN'T TAKE A ";W$:
     RETURN
1330 IF F(KW)=3 THEN PRINT "YOU
     HAVE ALREADY GOT IT."
1340 LET F(KW)=3: RETURN
1350 IF GE=0 OR GE=15 OR GE=23
     THEN PRINT "GIVE WHAT TO
     WHOM?": RETURN
1360 IF F(KW)<>3 THEN PRINT "YOU
     HAVEN'T GOT THE ";W$: RETURN
1370 IF GE>12 AND KW>2 AND KW<9
     THEN GO TO 1390
1380 IF GE=11 AND KW<>10 THEN
     LET F(KW)=0: RETURN
1390 LET F(GE)=0: LET F(GE+1)=3:
     PRINT "THE ";O$(GE);" SAYS
     'THANK YOU'"
1395 PRINT "HE GIVES YOU THE ";
     O$(GE+1)'"AND GOES AWAY.":
     RETURN
1400 IF PL<>4 OR PC<>5 THEN
     PRINT "THAT'S NO USE HERE.":
     RETURN
1410 IF KW<>14 THEN PRINT "IT'S
     NO USE SAYING ";W$: RETURN
1420 IF F(14)<>3 THEN PRINT "YOU
     DON'T KNOW THE WORD."
1430 LET F(26)=1: PRINT "A DOOR
     OPENS IN THE EAST WALL":
     RETURN
1450 IF PC<>5 OR F(25)<1 THEN GO
     TO 1490
```

```
1460 IF PL=6 AND F(2)=3 THEN LET
     F(27)=1: PRINT "THE DOOR IS
     OPEN": RETURN
1470 IF PL=6 AND F(2)<>3 THEN
     PRINT "YOU HAVEN'T GOT THE
     KEY.": RETURN
1480 IF PL=4 AND F(27)<>1 THEN
     PRINT "YOU HAVEN'T SAID THE
     WORD.": RETURN
1490 PRINT "WHAT DOOR?": RETURN
1500 IF KW=10 AND F(10)=3 THEN
     LET F(10)=1: PRINT "YUM
     YUM": RETURN
1510 IF KW=21 AND GE=21 THEN
     GO TO 1540
1520 IF F(KW)<>3 THEN PRINT "IT
     ISN'T YOURS TO EAT.": RETURN
1530 PRINT "YUK!": LET F(KW)=0:
     RETURN
1540 LET F(22)=1: LET F(21)=0:
     PRINT "BURP! SEE WHAT YOU'VE
     FOUND.": RETURN
1550 IF KW<>22 THEN PRINT "YOU
     CAN'T LIGHT A ";W$: RETURN
1560 IF F(8)<>3 THEN PRINT "YOU
     NEED MATCHES.": RETURN
1570 IF F(22)<>3 THEN PRINT "YOU
     NEED THE TORCH.": RETURN
1580 LET F(25)=1: PRINT "IT'S
     ALIGHT, BUT NOT FOR LONG.":
     RETURN
1600 IF F(20)<>3 THEN PRINT "NO
     AXE.NO CHOP!": RETURN
1610 IF KW<>1 THEN PRINT "YOU
     CAN'T CHOP DOWN ";W$: RETURN
1620 LET F(25)=1: PRINT "NOW YOU
     CAN SEE.": RETURN
1650 IF PL=8 AND (PC=6 OR PC=7)
     AND F(7)=3 THEN GO TO 1680
1660 IF F(7)<>3 THEN PRINT "YOU
     NEED A BOAT.": RETURN
```



```
1670 PRINT "CROSS WHAT? CROSS
     COMPUTER SOON!": RETURN
1680 LET PC=7-(PC=7): PRINT
     "SAFELY ACROSS.": RETURN
1700 IF KW<>3 AND KW<>12 AND
     KW<>20 THEN PRINT "YOU
     CAN'T KILL WITH THAT.":
     RETURN
1710 IF GE=0 THEN PRINT "KILL
     WHAT?WHO?": RETURN
1720 IF GE=11 OR GE=13 OR GE=17
     OR GE=19 THEN GO TO 1760
1730 IF GE=15 AND KW=12 THEN
     GO TO 1780
1740 IF RND>.6 THEN PRINT
     "MISSED!": LET EU=1: RETURN
1750 PRINT "KILLED HIM!": LET
     O$(GE)=" ": RETURN
1760 LET F(GE)=0: PRINT "A BIT
     UNNECESSARY, BUT HE'S DEAD.":
     RETURN
1780 PRINT "IT'S DEAD, AND YOU'VE
     FOUND THE LOST CROWN OF
     UMBIMAJINI.": GO TO 1070
1999 RETURN
2000 REM "INITIALIZATION"
2010 DIM R$(8,8,14): DIM E$(8,8,4):
     DIM O$(30,11): DIM P(27,2):
     DIM V$(11,5): DIM F(27)
2020 FOR N=1 TO 8: FOR T=1 TO 8:
     READ R$(N,T),E$(N,T):
     NEXT T: NEXT N
2030 DATA "DESERT"," ", "DESERT","WS",
     "CLEARING","ES","JUNGLE","/EWS"
2035 DATA "SWAMP", " ","JUNGLE",
     "/EWS","PLAIN","WE","LOW HILL",
     "WS"
2040 DATA "DESERT","NE","DESERT","WNE",
     "PLAIN","NSW","RUIN","NE"
```

```
2045 DATA "JUNGLE","/ND",
     "CLIFF SIDE","UDN",
     "CLIFF PATH","UDS",
     "HILL SIDE","UN/S"
2050 DATA "PORT BATA","ES","OLD
     STREET","ESW","EDGE OF
     TOWN","NSW","PASSAGE","/ENS"
2055 DATA "STAIRS","/UD","CAVE","E/W",
     "CLIFF PATH","NSW","HUT","N"
2060 DATA "BEACH","NES","DUNES",
     "NSW","SWAMP"," ","SMALL ALCOVE",
     "/NE"
2065 DATA "TEMPLE ROOM","/UWE",
     "INNER SANCTUM","W",
     "CLIFF TOP","DNS","SWAMP"," "
2070 DATA "QUICKSAND"," ","PLAIN",
     "NESW","JUNGLE","/NEW",
     "PLANTATION","WS"
2075 DATA "ENTRANCE HALL","S/ED",
     "PRIEST'S ROOM","/W","CLIFF
     PATH","NES","ROCKY LEDGE","UDW"
2080 DATA "BEACH","NES","UBIMBI
     VILLAGE","NES","SHOP","WS",
     "JUNGLE","NES"
2085 DATA "JUNGLE","/NEW",
     "CLEARING","WE","MOUNTAIN",
     "DSW","MOUNTAIN PASS","DS"
2090 DATA "BEACH","NS","STREET",
     "NS","YARD","N","JUNGLE",
     "/NES"
2095 DATA "PLAIN","ESW","LAKE","EW",
     "LARGE NEST","NESW","JUNGLE",
     "/WN"
2100 DATA "JUNGLE","/NE","VILLAGE
     END","NEW","PLAIN","EW",
     "PLAIN","NEW"
2105 DATA "JUNGLE","/NEW","WEST
     BANK","W","EAST BANK",
     "NE","WEETU VILLAGE","W"
```



```
2110 FOR N=1 TO 30: READ O$(N):
     IF N<28 THEN READ
     P(N,1),P(N,2),F(N)
2120 NEXT N
2130 DATA "JUNGLE",0,0,1,"KEY",
     2,4,0,"GUN",3,1,1,"BEADS",
     7,2,1,"CLOTH",3,3,1
2140 DATA "MONEY",3,1,1,"BOAT",
     4,1,1,"MATCHES",3,2,1,
     "RIVER",8,6,1
2150 DATA "BANANAS",5,4,1,
     "GORILLA",7,7,1,"STICK",
     7,7,0
2160 DATA "HERMIT",3,8,1,"WORD",
     3,8,0,"SNAKE",4,6,1,
     "CROWN",4,6,0
2170 DATA "WITCHDOCTOR",6,2,1,
     "MEDICINE",6,2,0,
     "SHOPKEEPER",6,3,1,
     "AXE",6,3,0
2180 DATA "CANNIBAL",8,8,1,
     "TORCH",8,8,0,"LION",6,6,1,
     "DOOR",6,5,0,"U",0,0,0,"D",
     0,0,0,"N",0,0,0,"E","S","W"
2190 FOR N=1 TO 11: READ V$(N):
     NEXT N
2200 DATA "LOOK","GO","TAKE","GIVE",
     "SAY","OPEN","EAT","LIGHT",
     "CHOP","CROSS","KILL"
2210 LET PL=3: LET PC=1: LET
     EU=0
2220 RETURN
3000 IF LEN D$<5 THEN LET
     D$=D$+" ": GO TO 3000
3010 RETURN
3020 IF LEN W$<11 THEN LET
     W$=W$+" ": GO TO 3020
3030 RETURN
```

# 11

# Game design

You can always be sure of one thing with an Interactive game. If it takes a month to write the program, it will take two months to debug it! In both action and adventure games, the player's control of the game is fairly limited – there are only certain keys which he can press, or certain commands which can be entered. It's not too hard to work out all the things which can happen in the game, and to make sure that your program caters for them all. Interactive games offer far more opportunities for things to go wrong. Not only is it hard to see all the ways in which the player could tackle the game, it's also hard to predict what decisions the computer will make.

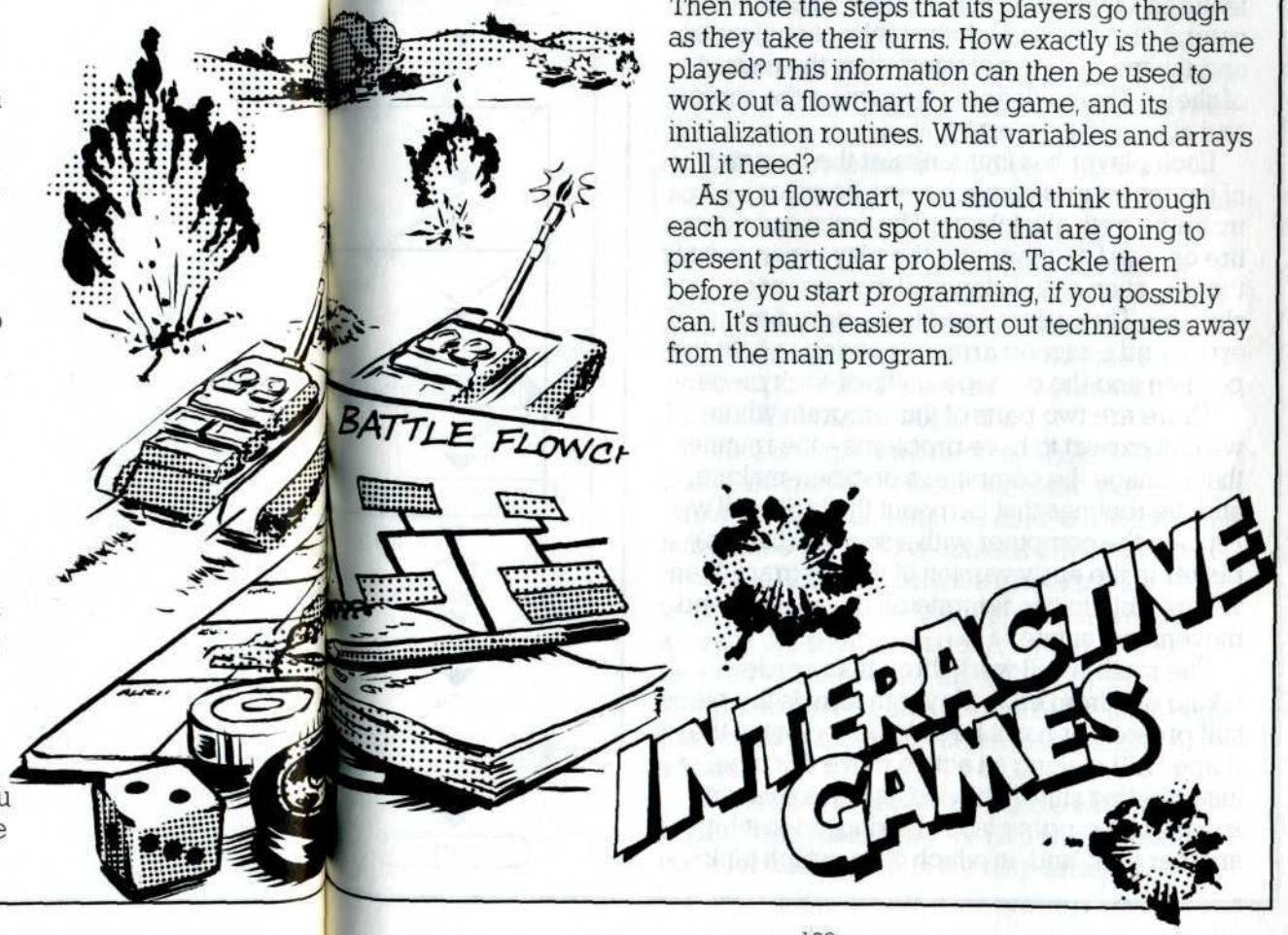
This section concentrates on one simple interactive game – BATTLE – and looks at how it was designed and written.

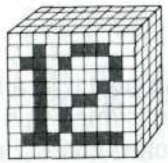
The object of the game is to simulate a tank battle, where the two opposing forces can manoeuvre for position and slog it out in exchanges of fire. There are a number of board games around based on the same general idea, and one of those formed the particular inspiration for this program.

Whatever the style or type of game that you want to write, the first step is to have a clear idea of its essence. What kinds of images do you want the game to conjure up for its players? The second stage is to develop the game on paper, and to play it as a board game, or a pencil and paper game. This is the best way to work out the rules, and to find out what size of board and what number and variety of pieces you need to make a good game. If you are going to base your program on an existing game, then sit down and play that game until you know it inside out.

The last, and longest, stage is to convert the game to a working program. Start by writing down a description of the game, and its rules. Then note the steps that its players go through as they take their turns. How exactly is the game played? This information can then be used to work out a flowchart for the game, and its initialization routines. What variables and arrays will it need?

As you flowchart, you should think through each routine and spot those that are going to present particular problems. Tackle them before you start programming, if you possibly can. It's much easier to sort out techniques away from the main program.





# First moves

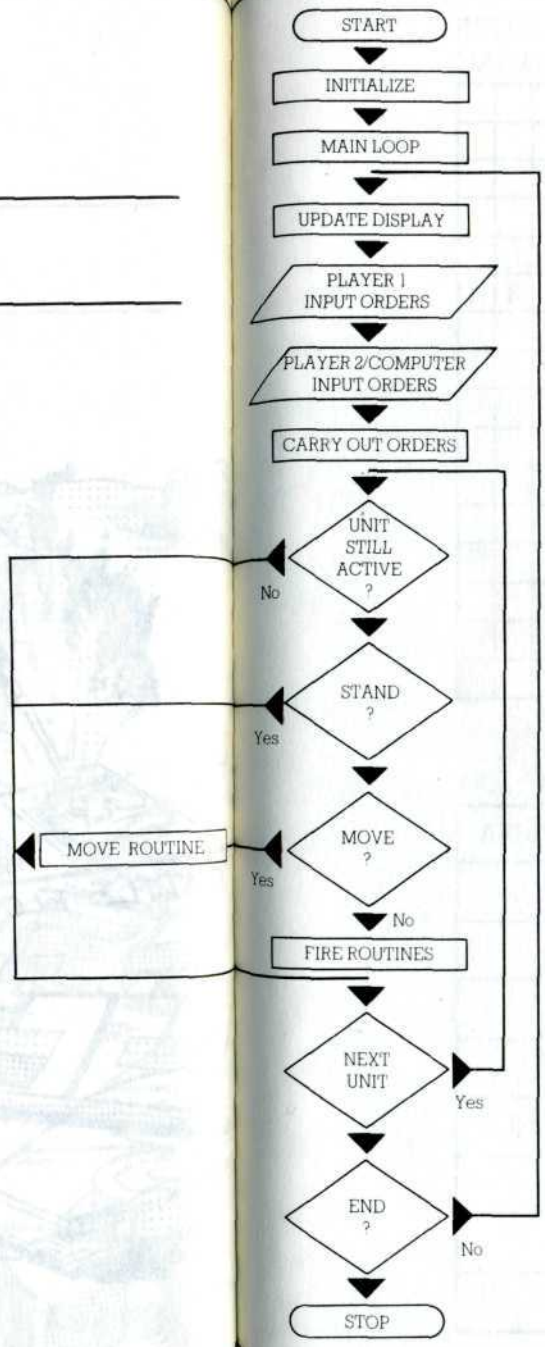
In BATTLE, the game is played on a 16×16 grid – the battlefield – on which are marked three features – two rivers and a wood. These are natural obstacles which the tanks cannot cross, and the wood does not even allow the passage of shells. The battlefield is shown on the screen, and also held in an array.

Each player has four tanks, at the beginning of the game, and on each move he gives instructions to all of these. The tanks can move, fire or stand inactive, and the player can specify the direction and distance of movement or shelling. The orders need to be stored in an array, and a second array is needed to hold the position and the damage status of each piece.

There are two parts of this program where we can expect to have problems – the routines that manage the computer's decision-making, and the routines that carry out the orders. If we replace the computer with a second (human) player in the early version of the program, then we are able to concentrate on the analysis and movement routine.

The routine will work through the orders, taking one from each player in turn. Is the piece still present, or has it been knocked out of the game? Is it making an active move this time, or just standing still? If moving, is there a vacant square in the right place? If firing, does it hit another tank, and, in which case, which tank?






How much damage does the hit cause? Enough to knock out the other tank?

The program lines from 300 on, and the subroutines at 1000 and 1200, manage this part of the program. The orders are held in the array O$(2,4,3) – two armies, four tanks in each, and up to three parts for each order – the type of command, and direction and distance where necessary.

Movement involves relatively little. The tank is removed from its old position both on screen and in the array, and its new position is then calculated from the given direction. If this square is already occupied by another tank, or by an obstruction, then the tank is replaced where it was. Otherwise, its new position is entered on the screen, in the array, and in the line and column variables of the tank's own array. The screen is laid out so that the line and column positions for the array agree with the PRINT AT lines and columns.

Firing is slightly more complicated. The shells are moved in much the same way as the tanks, and, in fact, both are able to use the same subroutine (at 1100) to convert direction into a change of line and column. Checking for a HIT is a simple matter of looking at what is in the array at the shell's square. A further routine is needed to analyse the hits (1200 on). This compares the line and column of the shell with the stored positions of all the tanks, until it finds a match.

This is not the only way to cope with this type of problem. An alternative is to use a separate code for each piece in the map array.

In BATTLE, the player's tanks are all coded as 'T', the computer's/second player's, as 'E' (for Enemy). '1234' and 'ABCD' could have been used instead. This would have made it very simple to spot which tank had been hit, but would have created other problems elsewhere in the program. Look up the lines where the computer checks for 'T' and 'E' and imagine how they would look under the alternative system.

How quickly should the game end? This is the key question when working out the damage assessment routine. If you are converting an existing game into a program, then you can follow the methods used in the game. In BATTLE, the amount of damage caused by a hit depends upon two factors – the distance between the tanks and a random number (line 1250). The damage status of the tank that is firing the shell could be worked into the equation if you felt it was useful. This line would do it:

```
LET DAMAGE = ((5-E)*INT(RND*10)+10)
  *A(P,N,3)/100
```

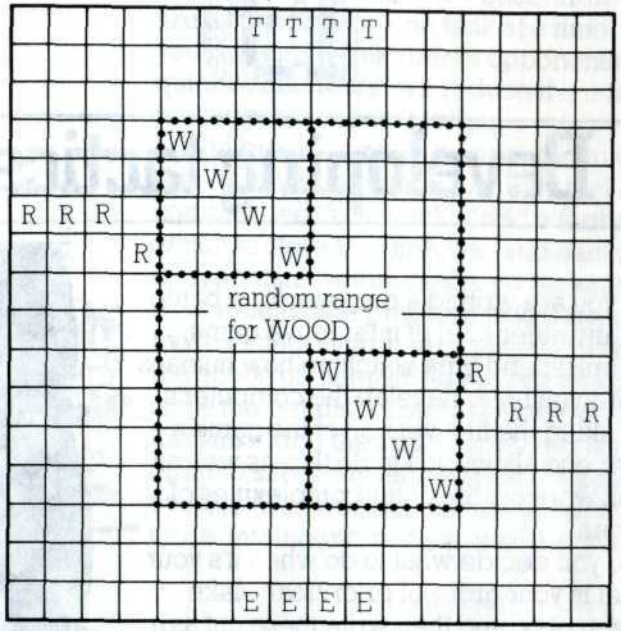
'E' is the variable from the loop that moves the shell. Its value is equivalent to the distance between the tanks.

A(P,N,3)/100 is the percentage damage status, where 100% means undamaged. Probably the best approach to damage calculations is to keep playing through the game with different damage lines, until you find one that gives the right effect.



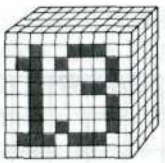
INITIAL POSITIONS

MAP ARRAY M$(16,16)

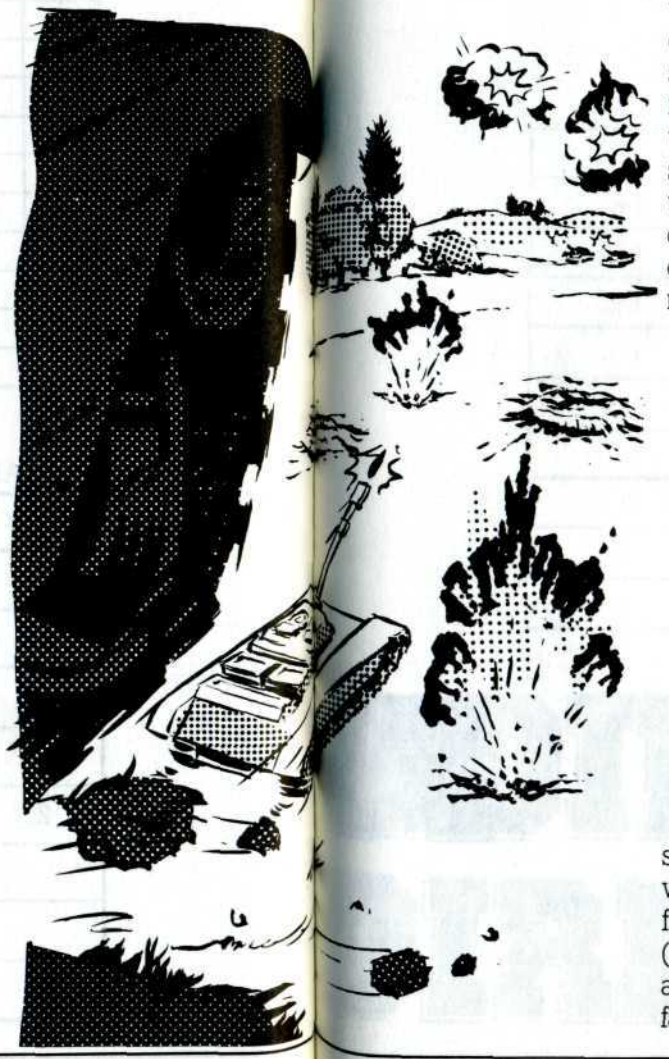



ARMY ARRAY A(2,4,3)

| ARMY | TANK | LINE | COLUMN | DAMAGE |
|---|---|---|---|---|
| 1 | 1 | 1 | 7 | 100 |
| | 2 | 1 | 8 | 100 |
| | 3 | 1 | 9 | 100 |
| | 4 | 1 | 10 | 100 |
| 2 | 1 | 16 | 7 | 100 |
| | 2 | 16 | 8 | 100 |
| | 3 | 16 | 9 | 100 |
| | 4 | 16 | 10 | 100 |



# Developing tactics



Whether you are writing a program for a battle game, or any other kind of interactive game, you must first spend time watching how humans play before you try to develop the computer's tactics. Making the first version of the game a two-player one allows you to do this, as well as letting you work out the other complexities of the program.

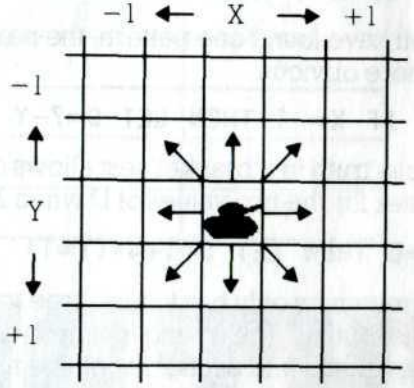
How do you decide what to do when it's your turn? What is your order of priorities? Make notes as you play, and then write these out as a set of game-playing rules. Next, play the game again, several times, and follow your own rules exactly. Do they work? At the very least, your rules should not let you miss any obvious opportunities and should not lead you into hopeless situations.

If you have got the BATTLE program typed in and running, then try playing it, on the two-player version, using the order of priorities that the computer follows. Note that the main attack line is calculated during the initialization stage, and the tanks advance up columns 5 and 6, or up columns 11 and 12 if the wood blocks the passage to the left.

The rule set is not particularly sophisticated, and there are a number of ways in which it could be improved. At the moment it shoots at the first target that it finds, rather than selecting the best target. The 'best' target is the tank which can be knocked out most quickly, and a development of this is to concentrate as much fire as possible on one tank at a time. That way you reduce the number of opponents most quickly, and therefore reduce the amount of damage that you will suffer.

The program could be adapted to include these tactics, though it obviously makes it more complicated. You would need a further array in which to store the direction and damage status of every tank within range. At the end of the scan routine, the program would then select the most damaged tank as its target. To coordinate the action of different tanks, you would need to store the details of every possible target of all the tanks, and then compare them with each other to see if any tank turned up in more than one set. This would make the program much more 'intelligent', but also much slower.

*The Scan*




The same scan routine is used when a tank is seeing if any enemy tanks are in range, and when it is looking for a nearby enemy. The lines from 3200 check along each possible line of fire (or movement) in turn, working from left to right, and top to bottom. The variable H governs how far it should check.

There was an interesting little problem involved in finding a single direction value – for the computer's orders – from the scan routine. This produces X and Y values of either −1,0 or +1. You could use eight separate lines, like this:

```
IF X=-1 AND Y=-1 THEN LET D=8
IF X=0 AND Y=-1 THEN LET D=1
```

etc, etc

The routine would work, but it's long, and it will slow the program down. The computer will have to check a total of sixteen possibilities as it works through those lines.

To find a more compact way of doing this, we have to look for a pattern in the values of X, Y and D.

The table shows that D will be 2,3 or 4 when X=+1, and that it will be 2 when Y=−1,3 when Y=0 and 4 when Y=+1. This leads us to the line:

```
3500 IF X=1 THEN LET D=3+Y
```

When you have found one pattern, the next is always more obvious.

```
3510 IF X=-1 THEN LET D=7-Y
```

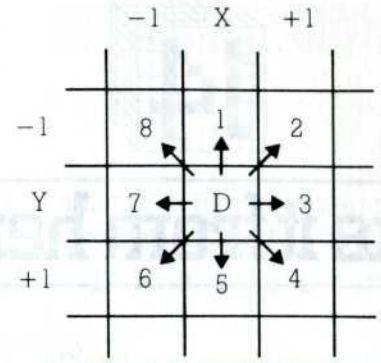
A simple 'truth in a bracket' test allows one line to cater for the two values of D when X=0:

```
IF X=0 THEN LET D=1+4*(Y=1)
```

The program now only has to test three values of X in this routine. The saving in time is obvious.

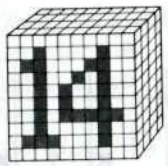
Whenever you find a long, repetitive routine, look for ways of compacting. Usually the increase in speed is worth the effort of rewriting the lines, but not always. Sometimes the more compact routine will be so involved and complex that it is not worth using – you, after all, will be faced with the problem of debugging it, and will you remember exactly how it works in a few months' time?






| X | Y | D |
|---|---|---|
| +1 | −1 | 2 |
| +1 | 0 | 3 |
| +1 | +1 | 4 |
| −1 | −1 | 8 |
| −1 | 0 | 7 |
| −1 | +1 | 6 |
| 0 | −1 | 1 |
| 0 | 0 | — |
| 0 | +1 | 5 |

# FINDING THE DIRECTION



# Take it from here

## Strategies

There's room for improvement in the BATTLE program, in both meanings of the term. The program could definitely be improved, and there's memory space in which to write your extensions to the program.

Ways of improving the computer's tactics have already been covered, but what about its strategy? At the moment its master plan is to find where the wood isn't and advance. This approach is hardly likely to give a hard time to any Napoleons among its human opponents, but could it be improved? The answer must be yes.

How does a thoughtful human (you) approach the game? What possible opening moves can you make? Is it best to attack along a single broad front, or does it pay to split your force? How does the position of the wood affect the game? In what ways do you respond to your opponent's plan of campaign?

As you find answers to these questions, write them down in terms of orders of priorities, or of specific responses to specific situations. They can then be added into the program. For example, if you wanted the computer's tanks to advance four abreast, if there was room, you would start by including this line in the initialization routine after the wood's position has been set. It checks that columns 5 to 8 are clear.

```
IF C>8 THEN LET P1=1
```

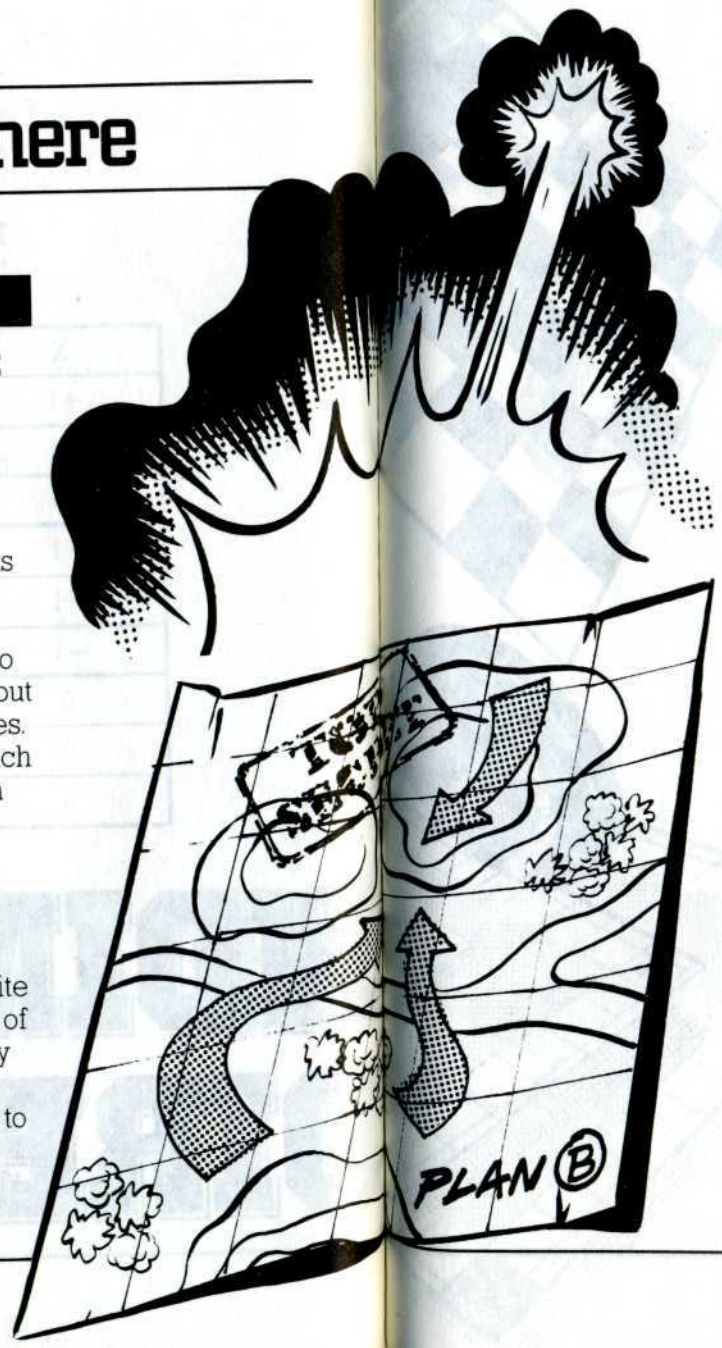
(C = wood column)

Plan 1 is now operational. It is worked into the computer's move by a new line:

```
3065  IF P1=1 AND C>=5 AND C<=8
      THEN GOTO 3100
```

(C = tank column)

Plan 2 would use a similar pair of lines to find and move up a broad column on the other side of the wood.




## Landscaping

Your map could include other features such as hills and roads. Hills would slow down the tanks, but offer improved shelling distances from the summit. Roads could give greater speed. If you feel the map is too small to accommodate all the features you would like to include, then use a larger map. You could reduce the amount of other information on the screen, or remove it altogether, writing in a new 'Help' page that the player could call up at will.

## Other units

A whole new range of possibilities, and complexities, can be introduced into your game by including other units such as artillery or anti-tank infantry units. These would have different speeds of movement, ranges of fire, and resistance to hits. Each different type would need a separate set of control routines, but they would follow the same general plan as those used for the tanks.

For a more mobile game, you could try 'redesigning' your tanks, so that they can fire while they are moving. If you restrict the firing to the direction of movement, then this won't raise any major problems.

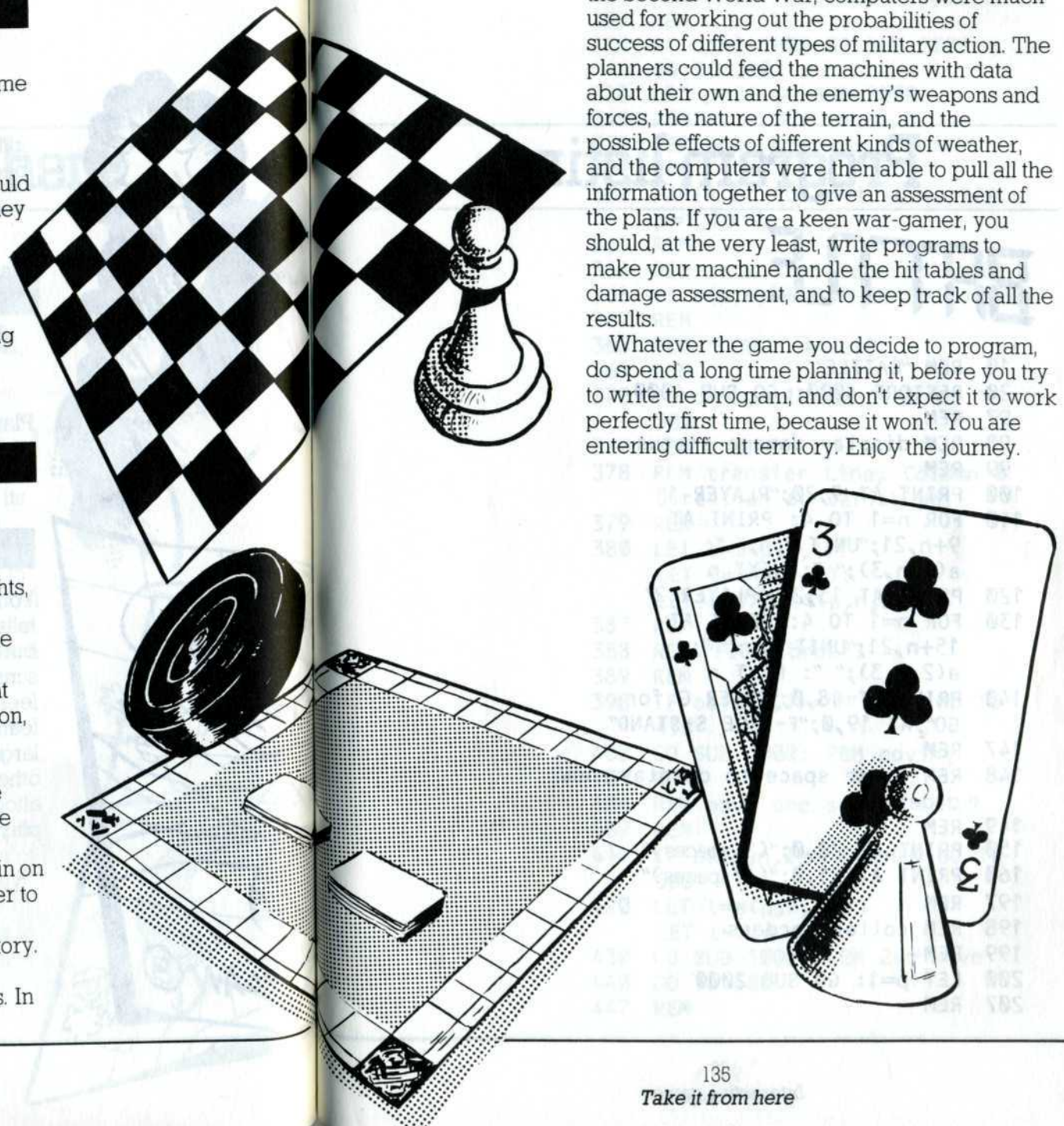
## A game of your own

What will your interactive games be? Most board games can be programmed, though some will take more thought and effort than others. The tactical games like chess, draughts, Chinese Checkers and Ludo all offer real challenges. (The major challenge on Chinese Checkers is how you cope with a hexagonal board!) Other board games present different problems of programming and of presentation, but the routines to work out the computer's decisions are usually quite simple.

Card games can be programmed and are particularly interesting if you are keen on the mathematics of probability. At the time of writing, card-playing programs are fairly thin on the ground – perhaps because people prefer to play cards with people.

War-gaming is traditional computer territory. Some of the first computers ever built were used for calculating the trajectories of shells. In the Second World War, computers were much used for working out the probabilities of success of different types of military action. The planners could feed the machines with data about their own and the enemy's weapons and forces, the nature of the terrain, and the possible effects of different kinds of weather, and the computers were then able to pull all the information together to give an assessment of the plans. If you are a keen war-gamer, you should, at the very least, write programs to make your machine handle the hit tables and damage assessment, and to keep track of all the results.

Whatever the game you decide to program, do spend a long time planning it, before you try to write the program, and don't expect it to work perfectly first time, because it won't. You are entering difficult territory. Enjoy the journey.



# Program listing

# BATTLE

```
 10 REM "BATTLE"
 20 RESTORE 4000: GO SUB 4000
 97 REM
 98 REM display damage status
 99 REM
100 PRINT AT 9,20;"PLAYER 1"
110 FOR n=1 TO 4: PRINT AT
    9+n,21;"UNIT ";n;" ";
    a(1,n,3);" ": NEXT n
120 PRINT AT 15,20;"PLAYER 2"
130 FOR n=1 TO 4: PRINT AT
    15+n,21;"UNIT ";n;" ";
    a(2,n,3);" ": NEXT n
140 PRINT AT 18,0;"ENTER G for
    GO";AT 19,0;"F-FIRE S-STAND"
147 REM
148 REM clear space to display  new
    orders
149 REM
150 PRINT AT 20,0;"(32 spaces)"
160 PRINT AT 21,0;"(32 spaces)"
197 REM
198 REM collect orders
199 REM
200 LET p=1: GO SUB 2000
207 REM
```



```
208 REM cp=1 if computer plays
209 REM
210 IF cp=1 THEN GO SUB 3000:
    GO TO 350
220 LET p=2: GO SUB 2000
349 REM
350 REM analysis and movement
351 REM
360 PAPER 8: FOR n=1 TO 4: FOR
    p=1 TO 2
363 REM
364 REM tank still operational?
365 IF a(p,n,3)=0 THEN GO TO 500
367 REM
368 REM "STAND" order?
369 REM
370 IF o$(p,n,1)="S" THEN GO TO
    500
377 REM
378 REM transfer Line, Column &
    Direction to variables
379 REM
380 LET l=a(p,n,1):
    LET c=a(p,n,2):
    LET d=VAL o$(p,n,2)
387 REM
388 REM "FIRE" order?
389 REM
390 IF o$(p,n,1)="F" THEN GO TO
    450
400 GO SUB 1000: REM moving
407 REM
408 REM move one square only?
409 REM
410 IF o$(p,n,3)="1" THEN GO TO
    500
420 LET l=a(p,n,1):
    LET c=a(p,n,2)
430 GO SUB 1000: REM 2nd move
440 GO TO 500
447 REM
```

```
448 REM fire routine
449 REM
450 FOR e=1 TO VAL o$(p,n,3):
    GO SUB 1100
460 IF m$(l,c)="w" THEN LET e=4:
    GO TO 490: REM in wood
467 REM
468 REM move shell
469 REM
470 LET z$=m$(l,c): PRINT AT
    l,c;"*":BEEP .02,0
477 REM
478 REM check for hit
479 REM
480 PRINT AT l,c;"B": IF z$="T"
    OR z$="E" THEN GO SUB 1200
490 NEXT e
500 NEXT p: NEXT n
507 REM
508 REM check for win
509 REM
510 FOR p=1 TO 2: LET v=0: FOR
    n=1 TO 4: LET v=v+a(p,n,3):
    NEXT n
520 IF v=0 THEN GO TO 600+
    (100 AND p=2)
530 NEXT p: GO TO 100
600 PRINT AT 20,1;"*** Red wins.
    ***": STOP
700 PRINT AT 20,1;"*** Blue
    wins. ***": STOP
1000 PRINT AT l,c;"B": LET
    m$(l,c)=" "
1010 GO SUB 1100: REM change line
    and column
1017 REM
1018 REM is move blocked?
1019 REM
1020 IF m$(l,c)<>" " THEN GO TO
     1040
```



```
1030 LET a(p,n,1)=l: LET
     a(p,n,2)=c
1037 REM
1038 REM display tank
1039 REM
1040 PRINT AT a(p,n,1),a(p,n,2);
     INK p;"A"
1050 LET t$="T": IF p=2 THEN LET
     t$="E"
1057 REM
1058 REM enter in array
1059 REM
1060 LET m$(a(p,n,1),a(p,n,2))=t$
1070 RETURN
1100 LET l=l+(d>3 AND d<7)-
     (d=8 OR d<3)
1110 LET c=c+(d>1 AND d<5)-(d>5)
1120 RETURN
1200 PRINT AT l,c; INK
     1+(z$="E");"A"
1219 REM
1220 REM which was hit?
1221 REM
1228 REM check line and column
1229 REM
1230 FOR x=1 TO 2: FOR y=1 TO 4:
     IF a(x,y,1)<>l OR
     a(x,y,2)<>c THEN GO TO 1300
1237 REM
1238 REM make tank flash
1239 REM
1240 LET z=22528+32*l+c: POKE z,
     (PEEK z+128)
1247 REM
1248 REM damage assessment
1249 REM
1250 LET damage=(5-e)*INT
     (RND*10)+10
1260 LET a(x,y,3)=a(x,y,3)-damage
1267 REM
1268 REM knock out if under 10
```

```
1270 IF a(x,y,3)>=10 THEN GO TO
     1290
1280 LET a(x,y,1)=0:LET a(x,y,2)=0:
     LET a(x,y,3)=0: PRINT AT
     l,c;"B": LET m$(l,c)=" ":
     LET y=4: LET x=2: GO TO
     1300
1287 REM
1288 REM turn off flash
1289 REM
1290 BEEP .1,25: POKE z,(PEEK
     z-128)
1300 NEXT y: NEXT x:
     LET e=4: RETURN
2000 REM get orders
2001 REM
2010 FOR n=1 TO 4
2017 REM
2018 REM tank still operational?
2020 IF a(p,n,3)=0 THEN GO TO
     2150
2027 REM
2028 REM make tank flash
2029 REM
2030 LET Z=22528+32*a(p,n,1)+
     a(p,n,2): POKE z,PEEK z+128
2035 REM
2038 REM collect order codes
2039 REM
2040 PRINT AT 19+p,n*6; INK p;n;:
     POKE 23658,8: INPUT AT
     0,0;("Player ";p;":Unit ";n;"
     - Move ?");a$
2050 IF a$="S" OR a$="F" OR a$="G"
     THEN GO TO 2070
2057 REM
2058 REM ignore invalid inputs
2059 REM
2060 GO TO 2040
2067 REM
2068 REM transfer to array
```



```
2069 REM
2070 PRINT a$;: LET o$(p,n,1)=a$:
     IF a$="S" THEN GO TO 2140
2080 INPUT AT 0,0;"Which
     direction? ";a$
2090 IF a$<"1" OR a$>"8" THEN GO
     TO 2080
2100 PRINT a$;: LET o$(p,n,2)=a$
2110 INPUT AT 0,0;"Distance ?";a$
2120 IF a$<"1" OR a$>"4" THEN GO
     TO 2110
2130 PRINT a$;: LET o$(p,n,3)=a$
2140 POKE z,PEEK z-128
2150 NEXT n: RETURN
2999 REM
3000 REM computer gives orders
3001 REM
3005 FOR n=1 TO 4: LET m=0: LET
     o$(2,n)=" "
3007 REM
3008 REM tank still operational?
3010 IF a(2,n,3)=0 THEN GO TO
     3190
3020 LET l=a(2,n,1): LET
     c=a(2,n,2)
3027 REM
3028 REM scan for enemy in range
3030 LET h=4: GO SUB 3200
3040 IF m=1 THEN GO TO 3190
3047 REM
3048 REM scan for enemy just out
     of range
3050 LET h=6: GO SUB 3200
3060 IF m=1 THEN GO TO 3190
3067 REM
3068 REM main line of advance
3069 REM
3070 IF c=ac OR c=ac+1 THEN GO
     TO 3100
3080 IF c<ac THEN GO TO 3110
3090 IF c>ac THEN GO TO 3140
```

```
3100 IF m$(l-1,c)=" " THEN LET
     o$(2,n,2)="1": GO TO 3180
3110 IF m$(l-1,c+1)=" " THEN LET
     o$(2,n,2)="2": GO TO 3180
3120 IF m$(l,c+1)=" " THEN LET
     o$(2,n,2)="3": GO TO 3180
3130 LET o$(2,n,1)="S": GO TO
     3190
3140 IF m$(l-1,c-1)=" " THEN LET
     o$(2,n,2)="8": GO TO 3180
3150 IF m$(l,c-1)=" " THEN LET
     o$(2,n,2)="7": GO TO 3180
3160 LET o$(2,n,1)="S": GO TO
     3190
3165 REM
3170 REM STAND if no good move
3175 REM
3180 LET o$(2,n,1)="G": LET
     o$(2,n,3)="2"
3190 PRINT AT 21,n*6; INK 2;n;
     o$(2,n): NEXT n: RETURN
3200 REM
3201 REM scan routine
3202 REM
3205 FOR y=-1 TO 1: FOR x=-1 TO
     1: FOR z=1 TO h
3210 IF x=0 AND y=0 THEN GO TO
     3260
3220 LET l1=l+z*y: LET c1=c+z*x
3230 IF c1<1 OR l1<1 OR c1>16 OR
     l1>16 THEN LET z=8: GO TO
     3260
3237 REM
3238 REM spotted a target
3239 REM
3240 IF m$(l1,c1)="T" THEN GO SUB
     3500: GO TO 3260
3247 REM
3248 REM path blocked
3249 REM
```



```
3250 IF m$(l1,c1)<>" " THEN LET
     z=H
3260 NEXT z: NEXT x: NEXT y
3270 RETURN
3497 REM
3498 REM get Direction from scan
3499 REM
3500 IF x=1 THEN LET d=3+y
3510 IF x=-1 THEN LET d=7-y
3520 IF x=0 THEN LET d=1+4*(y=1)
3527 REM
3528 REM finish off order
3529 REM
3530 LET q$="G": IF h=4 THEN LET
     q$="F"
3540 LET o$(2,n,1)=q$: LET
     o$(2,n,2)=STR$ d: LET
     o$(2,n,3)="4"
3547 REM
3548 REM close loops
3549 REM
3550 LET z=8: LET x=1: LET y=1:
     LET m=1: RETURN
3999 STOP
4000 PAPER 7: INK 0: BORDER 7:
     CLS : PRINT AT 0,13;"BATTLE"
4010 PRINT AT 2,1;"Each move
     opens with you giving""each
     piece its orders."
4020 PRINT "The second player (or
     Spectrum)""gives his orders,
     and the moves"
4030 PRINT "are then played
     out.""POSSIBLE MOVES:"
4040 PRINT " GO.. FIRE.. STAND
     (do nothing)""Direction and
     distance must be"
4050 PRINT "given for GO or FIRE
     commands."
4060 PRINT " GO 1 or 2 squares."
```

```
4070 PRINT "Rivers & woods can't
     be crossed."""FIRE - Max. 4
     squares - more damage at
     close range."
4500 FOR n=1 TO 2: READ g$: REM
     graphics
4510 FOR r=0 TO 7: READ b: POKE
     USR g$+r,b: NEXT r: NEXT n
4520 DATA "A",0,0,31,24,126,255,
     126,60: REM tank
4530 DATA "B",0,1,0,1,0,1,0,85:
     REM grid lines
5000 DIM m$(16,16): REM map
5010 DIM o$(2,4,3): REM orders
5020 LET m$(6, TO 3)="rrr":
     LET m$(7,4)="r":
     LET m$(10,13)="r":
     LET m$(11,14 TO)="rrr"
5030 REM rivers on map
5040 LET l=INT(RND*6)+4: LET
     c=INT (RND*5)+5
5050 FOR n=l TO l+3: LET
     m$(n,c TO c+3)="wwww": NEXT
     n
5055 REM
5060 REM wood in random position
5065 REM
5070 LET ac=5: IF c<8 THEN LET
     ac=11
5075 REM
5080 REM attack line misses wood
5085 REM
5090 LET m$(1,7 TO 10)="TTTT":
     LET m$(16,7 TO 10)="EEEE"
5095 REM
5100 REM starting positions
5105 REM
5110 DIM a(2,4,3): REM 2 armies -
     4 pieces in each - line,
     column and damage status
5115 REM
```



```
5120 FOR n=1 TO 2: FOR e=1 TO 4:
     LET a(n,e,1)=1+(15 AND n=2):
     REM lines
5130 LET a(n,e,2)=e+6: REM
     columns of pieces
5140 LET a(n,e,3)=100: REM 100% =
     undamaged
5150 NEXT e: NEXT n
5160 BEEP 1,0: INPUT AT 0,0;"ONE
     PLAYER OR TWO ?"; LINE A$
5170 IF A$<>"1" AND A$<>"2" THEN
     GO TO 5160
5180 LET cp=VAL A$(1)
5200 BORDER 5: CLS
5205 REM
5210 REM blank map
5215 REM
5220 PRINT AT 0,0; PAPER 2;"(18
     spaces) "
5230 FOR n=1 TO 16: PRINT AT
     n,0; PAPER 2;" "; PAPER 7;
     "BBBBBBBBBBBBBBBB"; PAPER 2;
     " ": NEXT n : REM GRID
5240 PRINT AT 17,0; PAPER 2;"(18 spaces)"
5250 REM direction display
5255 REM USE KEYBOARD FOR SYMBOLS
     IN 5260
5260 PRINT AT 1,20;"DIRECTIONS";
     AT 3,22;"8 1 2";AT 4,23;
     "\|/";AT 5,22; "7-*-3"; AT
     6,23;"/|\";AT 7,22;"6 5 4"
5270 REM
5280 REM put details on map
5290 REM
5300 FOR l=1 TO 16: FOR c=1 TO
     16: IF m$(l,c)=" " THEN GO
     TO 5350
5310 REM rivers
5320 IF m$(l,c)="r" THEN POKE
     22528+32*l+c,40
5330 REM woods
```

```
5340  IF m$(l,c)="w" THEN POKE
      22528+32*l+c,32
5350  NEXT c: NEXT l
5360  REM display armies
5370  PRINT AT 1,7; INK 1;"AAAA"
5380  PRINT AT 16,7; INK 2;"AAAA"
5390  RETURN
```



# Appendix A

# Appendix A

## Essential BASIC

**ABS** gives the ABSolute value of a number by stripping off any minus sign.

**AND** compares two statements to see if they are both true.

Can also be used like this: LET X=X+(4 AND A$="8")

Here AND really means 'IF the following statement is true'.

**ATTR** gives the colour code for a square on the screen. Use it like this: ATTR(L,C)

**BEEP** produces a sound. BEEP is followed by two numbers e.g. BEEP 1,0. The first fixes the length of the note (1 = 1 second); the second controls the pitch (0 = middle C).

**BIN** used when you want to give a binary number for graphics definition. BIN 10000100 = 132.

**BORDER** sets the border colour, using the numbers 0 to 7. BORDER 2 for a red border.

**BRIGHT 1** makes things appear extra bright. BRIGHT 0 puts them back to normal.

**CHR$** converts an ASCII code number into a character, graphic, or control command.

**CIRCLE** draws a circle. Follow it by three numbers – the x and y co-ordinates of the centre, and the radius.



**CLEAR** clears the memory beyond the BASIC program, wiping out all variables, arrays, etc.

**CLEAR 50000** resets RAMTOP, the end of your BASIC program area, to address 50000 (48k machine.) Used to create a safe space in which to store machine code programs.

**CLS** clears the screen only. Other parts of memory are not affected.

**COPY** can only be used with a printer. It copies the screen onto paper. If the copy looks odd, it is because the printer only picks up those parts of the screen that are INKed.

**DIM** sets up an array. Initially all stores are empty, if it is a string array, or set to 0 if it is a number array. The DIM line must give the array name, and the dimensions of the array. DIM S$(20,20), DIM N(10,15,20), DIM X(12).

The subscripts of the array start from '1' and go up to the dimensions given. An array can be DIMensioned several times during a program – it's quite a useful way of clearing it for a fresh start.

**DRAW** will draw a line from the current PLOT position to a point given by x and y vectors.

**FLASH** is used in the same way as BRIGHT. FLASH 1 makes things flash. FLASH 0 is back to normal.

**FOR. . . .TO. . . .STEP. . .** is the first line of a FOR. .NEXT. . loop. This sets the range of numbers through which the loop must run. STEP is optional and can be missed out from a simple series (1,2,3,4, . . . .) FOR X= 1 TO 21 STEP 4: PRINT X: NEXT X would print 1,5,9,13,17,21.

**GOSUB** (line number) sends the program to a subroutine, storing the current line number in the gosub stack. When the program meets a RETURN command, it returns to the line at which it left the main program.

**GOTO** (line number) sends the program to the line given. If there is no line with the stated number, then the program will go to the next line down.

**IF....THEN....** test a condition, and the program branches if the condition is true.
**INK** is the colour in which characters are printed. Use a number between 0 and 7 to get the colours shown on those keys. INK 8 is transparent – characters will be printed in whatever INK colour happens to be set at the squares on which they appear. INK 9 will make the Spectrum print either in white, or black, whichever gives the best contrast with the PAPER colour at the print position.
**INKEY$** reads the keyboard. It can be used directly;

```
IF INKEY$ ="8" THEN X=X+2...
```

It is safer, however, to transfer the character read, from INKEY$ to a normal store. . . .

LET A$= INKEY$.
IF A$ ="8" THEN. . . .

**INPUT** waits for the user to type something in, and then to press ENTER to close the message. If you are friendly towards your users, you will usually include prompts in the INPUT lines, so that they know what they are supposed to do:

```
INPUT "PLEASE TYPE IN YOUR NAME"; N$
```

**INT** lops off the whole number part of any number. INT 4.567 = 4. INT 9.9999 = 9. Almost essential when using the RND function, as this produces decimals, and you will want whole numbers for most purposes.
**INVERSE** prints reversed characters by putting PAPER where the INK should be and vice versa.

**LEN** tells you the length of a string or a string



variable, i.e. how many characters in it.
**LET** gives a value to a variable. LET A=99. LET N$="PETER".
**LIST** displays the program on the screen.
**LIST 100** (or whatever) to start the list from a particular line.
**LOAD "MYGAME"** will search for a program called "MYGAME" and load it into the computer. LOAD"" will load the first program it finds.

**NEW** clears the BASIC area of memory, (the program and any variables and arrays) ready for a new program. Things beyond the BASIC area – UDG's and machine code routines – are not affected.
**NEXT** is the closing command for a FOR. .NEXT. .loop. As long as there is a number left, and it is within range of the STEP, then the program will loop back to the FOR.

**OR** tests to see if either or both of two statements are true.
**OVER 1** can produce some peculiar printed effects. Two characters can both appear at the same position, but where their INK dots coincide, the screen will be PAPER colour.
**OVER 0** is the normal condition.

**PAPER** is the colour of the background. Numbers 0 to 7 give the colours shown on the keys. PAPER 8 is transparent – anything printed this way will pick up whatever the background colour happens to be at that point. PAPER 9 is contrast, and will be either black or white, to give the best contrast to the current INK colour.
**PAUSE** makes the program wait for a given length of time. PAUSE 50 will cause a wait of 1 second. PAUSE 0 for an endless wait. The program can be moved on at any time by pressing a key. PAUSE 0: LET A$ =INKEY$ is one way to collect keystrokes.

**PEEK** (an address) will tell you the contents of any address. Some PEEKs are more useful than others.

**PLOT** puts a dot of INK at the point marked by x,y coordinates. PLOT 100,50 will plot a point 100 dots from the left, and 50 dots up.

**POKE** (an address) puts data into an address in memory. Machine code routines have to be POKEd into place.

**PRINT** puts things on the screen. The PRINT lines can contain a wide variety of commands to change the appearance of the printed material – colour commands are the most obvious.

**PRINT AT line, column;"...** will print at a given place on screen.

**PRINT SEPARATORS:** a semicolon (;) between two items will make the second item appear immediately after the first.

A comma (,) between two items makes the second item appear in the next available print zone.

An apostrophe (') between two items makes the second item appear on the next available line.

**PRINT TAB column;"...** will print at a given column on the next available line.

**PRINT, (ZONE).** A print zone is half a screen wide.

**RANDOMIZE** should be used at the start of any program where you are going to use random numbers. It makes sure that the random number sequence starts at a random place.

**READ** must be followed by a variable. It transfers DATA from the DATA line to the given variable.

**REM** marks the start of a line where you can write remarks to remind you what the program is doing. The computer ignores anything on a REM line.



**RESTORE** pushes the data marker back to the top of the data list. If you want to READ the same set of DATA several times – perhaps the data for your signature tune – then you must restore the data marker before it reads.

**RESTORE** followed by a line number sets the data marker to the data that starts at, or after, that line.

**RETURN** sends the program back from a subroutine to the main program.

**RND** produces a random decimal number between 0 and 1. It can be used as it is to set a random limit:

**RUN** clears all variables and starts the program. This can be replaced by GOTO followed by the first line number if the program has already been run and you want to keep the information that the variables contain.

**SAVE "MYGAME"** transfers a program to tape, and labels it "MYGAME".

**SAVE "MYGAME" CODE (number)** transfers a machine code program starting from the address given.

**SAVE"MYGAME" LINE 10** saves a named program so that when reloaded it automatically runs from line 10.

**SAVE "MYGAME" SCREEN$** transfers a screen to tape, with the label "MYGAME".

**SQR** gives you the square root of a number.

**STOP** halts the program. It can be restarted with CONT.

**STR$** converts a number to a string variable. STR$ 9 is "9".

**VAL** gives you the value of a number in a string, or string variable. VAL "99" = 99.

VAL will also give you the answer to a sum in a string. VAL " 3+4*SQR 16" = 19

**VERIFY** is used after SAVING a program, to make sure that the tape recording is good.

# Appendix B

## Defining your own graphics

The Spectrum has been designed so that it is easy to create your own graphics. An area of the character set has been left free to handle them, and there are two commands which allow a very simple transfer of a design from paper to memory.

When a character is printed on the screen, it occupies a grid of pixels (picture cells – each is a dot of light). The grid has eight rows, each of eight dots. The pattern for the character is held in the memory as a set of eight binary numbers. Whereas ordinary numbers use a range of ten different figures (0,1,2,3,4,5,6,7,8,9), binary numbers only use two figures, 0 and 1. In a computer like the Spectrum, the numbers are limited to a maximum size of eight digits, because this is the size of memory stores. Any address can hold one BYTE of information, and the BYTE consists of eight BITS – Binary digITS. If you could look into an address in the way that the computer does, then you would see something like this:

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

When the computer is displaying a character on the screen, it looks at the set of eight addresses where the pattern is stored, and prints a dot of



light for each BIT that is set to 1 in each row.

To define a character then, you need to put a new pattern of binary numbers into the memory space for that character. Spectrum's User Defined Graphics are to be found in Graphics Mode, on the letter keys 'A' through to 'U'. Here's how to define a jet into Graphics 'A'. Type it in, then PRINT (Graphics) "A".

```
10 POKE USR "A"+0,BIN 00000000
20 POKE USR "A"+1,BIN 11110000
30 POKE USR "A"+2,BIN 01101000
40 POKE USR "A"+3,BIN 01100100
50 POKE USR "A"+4,BIN 11111111
60 POKE USR "A"+5,BIN 01100100
70 POKE USR "A"+6,BIN 01101000
80 POKE USR "A"+7,BIN 11110000
```

POKE is the command used to put a number directly into a memory address. Normally you would have to give the numbers for these addresses, but not here.

USR 'A' tells the Spectrum that the address is that of the UDG on letter 'A'. There are, of course, eight addresses here, for the eight lines.

BIN is a special function which allows you to enter numbers in BINary form. You will find it in E mode on the letter key [B].

To redefine any other UDG, you could use the same basic program, changing only the USR letter, and the pattern. If you are going to be doing much in the way of graphics definition, you will soon find that this is a very slow way of going about it, and also that it eats up memory at an alarming rate.

The first stage in making the definition routines more compact is to use a loop, and READ in the binary numbers from DATA statements.

```
10 FOR R=0 TO 7: READ B
20 POKE USR "A" +R,B:NEXT R
```

```
30 DATA BIN 00000000, BIN
   11110000,BIN 01101000,
   BIN 01100100,BIN 11111111,
   BIN 01100100,BIN 01101000,
   BIN 11110000
```

The next stage is to learn to convert binary number patterns into normal decimal numbers. To do this, use the Conversion Table. Work through your number from left to right, and when you find a '1', note down the decimal number above that column. Add the decimal numbers together at the end. The DATA line in the example program can now be changed:

```
30 DATA 0,240,104,100,255,100,104,240
```

When you want to define more than one character – which will usually be the case – then you can compact even further by using a double loop. The outside loop determines the number of characters, and READs the letter for each character. The inside loop is the same as that used above.

```
10 FOR N=1 TO 2: READ G$
20 FOR R=0 TO 7: READ B
30 POKE USR G$+R,B: NEXT R: NEXT N
40 DATA "B",56,60,56,16,56,120,191,184
50 DATA "C",184,184,40,36,36,68,132,196
```

You will probably find it best, at first, to stick to the binary form of numbers, as it is then much easier to see how the numbers and the patterns relate to each other. Compact the DATA into decimal form later, when you are happy with the designs, and when you feel you need the extra memory space.

| Conversion table | | | | | | | |
|---|---|---|---|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |

Total = 64 + 32 + 8 = 104



# Appendix C

## Super screens

This section introduces two machine code routines that allow you to produce screen effects that you could never achieve in BASIC. When you are typing in these routines, or any other machine code routines for that matter, take extra care to ensure that you type exactly what is written. A mistake in a machine code routine will not make the Spectrum print a helpful Error Report. What normally happens is that the machine locks up and you have to pull the plug and start again.

You will get more out of machine code routines if you understand what they are supposed to do. Read up on Z80 machine code, and then use the Assembler codes in Appendix A of the Spectrum manual to analyse the routine.

### Screen Dump and Recall

This transfers an entire screen display to a reserved area in memory, and transfers it back again when wanted. The process is much the same as transferring screens to and from tape with SAVE and LOAD instructions, but infinitely quicker.

```
10 CLEAR 24999
20 FOR N=25000 TO 25023
30 READ X:POKE N,X: NEXT N
```

```
40  DATA 33,0,64,17,0,98
    1,0,27,237,176,201
50  DATA 33,0,98,17,0,64,
    1,0,27,237,176,201
```

Type it in and run it. Nothing visible has happened yet, but the code is in place. Now print something in the screen, then enter this:

```
RANDOMIZE USR 25000
```

Clear the screen, then enter this:

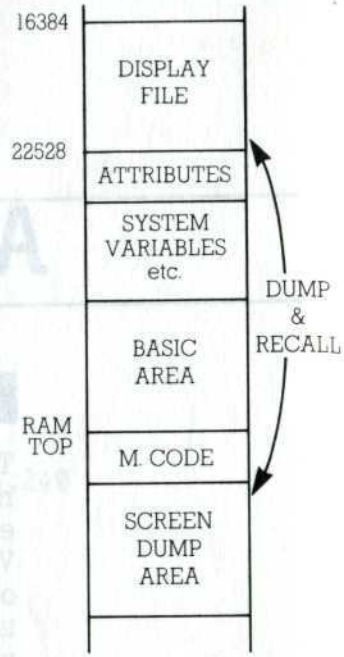
```
RANDOMIZE USR 25012
```

You should see your original screen again. RANDOMIZE USR 25000 sent the Spectrum to the machine code routine that started at address 25000. That routine – the data in line 40 - dumps the screen contents in another area of memory.

CLEAR 24999 resets RAMTOP to that address, to create an area beyond the BASIC program for your screen dump. This dump needs nearly 7k of memory, so you are only left with about 2k for your BASIC. If you have a 48k machine, this could be changed to CLEAR 56999. The machine code routine and screen dump area is now much higher in memory, and you have 34k left for your program. If you do this you must also change the next line to:

```
FOR N=57000 TO 57023
```

The machine code routine starts the screen dump at 25088. This needs moving to match the new RAMTOP. Change the 98's in the DATA lines to 223. 98*256=25088: 223*256=57088. This is the start of the new screen dump area.




### Using the Screen Dump

**1** Use screen dumps to save the game screen when you bring up a Help page. You could even hold the Help page in a dump, to allow



very slick transfer between game and Help. To do this you will need two dumps. This version of the routine will do it:

```
10  CLEAR 49999
20  FOR N=50000 TO 50047
30  READ X: POKE N,X : NEXT N
40  DATA 33,0,64,17,0,196,
    1,0,27,237,176,201
50  DATA 33,0,196,17,0,64,
    1,0,27,237,176,201
60  DATA 33,0,64,17,0,223,
    1,0,27,237,176,201
70  DATA 33,0,223,17,0,64,
    1,0,27,237,176,201
```

RANDOMIZE USR 50000, and RANDOMIZE USR 50012 dump and recall the first screen. RANDOMIZE USR 50024, and RANDOMIZE USR 50036 dump and recall the second screen.

**2** Use the routine for animating a graphic across a detailed background. Dump the background, print your graphic, and then recall the background to erase the graphic before you move it. It's fast, and very effective.

### Smooth Scrolls

Try this for a super title screen, or work it into an action game to give a moving background.

```
10  CLEAR 29999
20  FOR N=30000 TO 30016
30  READ X: POKE N,X: NEXT N
40  DATA 33,255,87,14,192,6,32,
    183,203,22,43,16,251,13,32,
    245,201
50  INK 6: PAPER 0: CLS
60  FOR N=1 TO 50: LET
    Y=INT(RND*176):
    LET X=INT(RND*256)
70  PLOT X,Y: NEXT N
80  LET Z=USR 30000
```

```
 90  LET Y=INT(RND*176):PLOT 255,Y
100  GOTO 80
```

'LET Z=USR 30000' is another way of calling up a machine code routine. If you use RANDOMIZE USR 30000 here, then you interfere with the random numbers, so that they are no longer random.

**Variations**

With a few minor alterations, you can make this routine scroll the screen from left to right. This program shows it in operation.

```
 10  CLEAR 29999
 20  FOR N=30000 TO 30016
 30  READ X: POKE N,X: NEXT N
 40  DATA 33,0,64,14,192,6,32,183,
     203,30,35,16,251,13,32,245,201
 50  LET L=0
 60  LET Z=USR 30000
 70  PRINT AT L,0;"■"
 80  LET L=L+1: IF L>21 THEN LET L=0
 90  GOTO 60
```

You don't have to scroll the whole screen. If you have watched a screen being loaded in from a tape, you will have noticed that it works in three blocks of eight lines each. Change DATA line to this if you only want the bottom third of the screen to move:

```
 40  DATA 33,0,80,14,64,6,32,183,
     203,30,35,16,251,13,32,245,201
```

The third number has been changed to 80, and the fifth to 64. These control where in the display file you start your scroll, and how much you move. There are three possible starting points – Top of screen (64), Middle section (72) and bottom third (80). To scroll a third of the screen only, the fifth number should be 64; 128 moves two-thirds of the screen, and 192 scrolls it all.

# Open up the fantastic world of computer games

## ZX Spectrum Game Master

Arcade games, adventure games, strategy games - they're all here in the ZX Spectrum Game Master.

**BUILD YOUR OWN GAMES**

Not just a book of listings but a guide to creating and personalising your own games. The ZX Spectrum Game Master provides all the building blocks you'll need to start programming your own games. Title pages, game screens, sound effects, movement routines, scoring routines. Everything you need and a lot more besides.

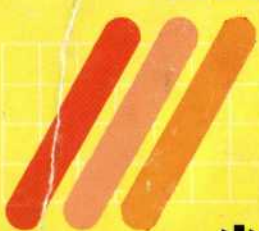
**FULL LISTINGS**

To start you off there are plenty of full length Spectrum games listed in the Game Master. Choose between arcade, adventure, and strategy games. Then, when you've played through a game, use the listings to develop something of your own. Put on your own title page - and your own name.
It's your game after all.

**GAME MASTER**

Move on to wholly original games. The tricks and routines in the ZX Spectrum Game Master will give you the confidence to set off on your own. How to develop your own ideas, how to plan your games, how to present them on screen, how to program them for maximum efficiency.

**MASTER THE ART OF THE SUPER GAMESTERS.**
ZX SPECTRUM GAME MASTER.



Longman
Computer
Books

ISBN 0-582-91606-2

9 780582 916067