TEXAS INSTRUMENTS
HOME COMPUTER

GAME WRITERS' PACK 1

CASSETTE SOFTWARE
WITH MANUAL

An integrated pack containing a series of programs on cassette that develop and graphically display major ideas covered in the accompanying book. Enables any user to progressively understand and make full use of this computer.

COLLINS
MICROSOFTWARE

TEXAS INSTRUMENTS
HOME COMPUTER

Game Writers' Pack 1

PK McBride

COLLINS
MICROSOFTWARE

Contents

Introduction

This Pack is the first of two that demonstrate the techniques and ideas needed for writing a wide variety of games in TI BASIC. Here we are dealing mainly with guessing games, on-screen action and maze-based adventure games. In Pack 2 you will discover how to tackle games of strategy that allow the computer to fight back.

The programs on the cassette are of two types. MAZE, RACETRACK and TARGET are working diagrams that demonstrate techniques in the simplest possible ways. These can be taken over by you and converted into fully fledged games if you wish. The other three programs, BAT, DRAGON and DUEL are given as examples of the types of games that can be written in TI BASIC using the ideas of this book.

TI BASIC was designed for simplicity, not speed, and you will find that screen action will always be rather slow compared to arcade games. If, when you have worked through the book, you find that you want to develop further with action games, then you will find it well worthwhile to get an EXTENDED BASIC cartridge. This will allow you to use SPRITES, which give a much faster and smoother movement. EXTENDED BASIC also has many other facilities for the advanced programmer. A brief outline of some of these is given in Appendix B at the end of this book.

The book assumes that you have a reasonable grasp of BASIC programming up to the level covered by the two Starter Packs – that is, just about all of the TI BASIC commands, statements and functions except for those used in file-handling. It also assumes that you possess no peripherals apart from the cassette leads. Use of the Joysticks is covered in the book, but all of the programs are designed to be useable even without them.

1
Take it from the top

So you are getting tired of playing other people's games and want to write your own! Why not. Games programming is great fun, and an excellent way of getting to grips with the mysteries of the computer. It can also have the useful spin-off of entertaining the other members of your family – the ones who have complained about the amount of time you spend locked up with the machine.

A good game need not be difficult to write. Some of the best use very simple ideas but have a top dressing of graphics and sound effects to turn them into amusing and original games. You will often find that the special effects take longer to write than the main program, but they are fiddly, rather than difficult, and the only real limitation is the scope of your own imagination.

There are essentially two ways of starting to write a game. You can begin with an effect that a BASIC routine produces and work this up into a game. The games arising from the CALL COLOR sub-routine that are given in the 'Colour changing' chapter are examples of these, and you will find many others eleswhere in the book.

The second approach is sometimes called 'Top-down' programming. Here you decide what the game is going to be about first, and then you find some way of turning it into a program. When you are working this way you should expect to spend a long time first with pencil and paper before you ever come to the computer. If you can write down exactly – and it must be exactly – what the program is supposed to do, using clear and simple English, then you should be able to write it in BASIC. You should also plan your screen layouts on squared paper, and work out the hex strings you need for your graphics characters before you reach the keyboard. It really makes life easier in the long run.

Don't miss out the flowchart stage. It's the best way to see how the program is supposed to work. You can start by sketching in the broadest outlines.

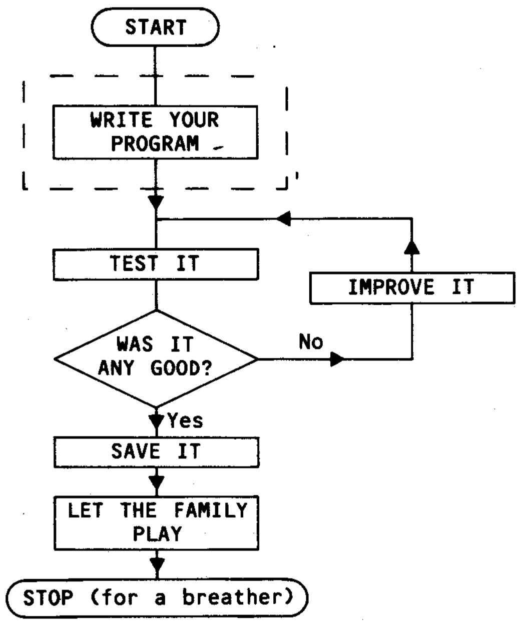


Figure 1

You can then start to expand the more complicated parts of the flowchart. What does it mean 'Write your program'?

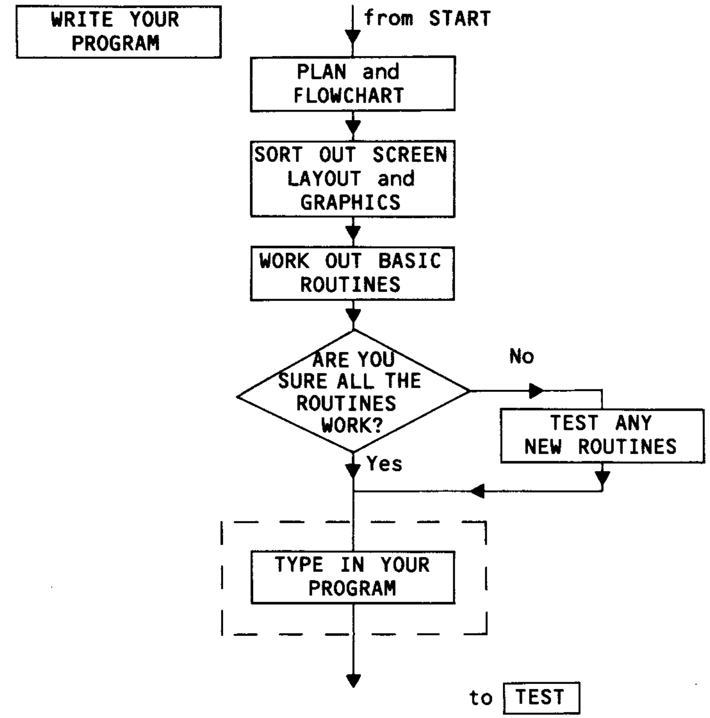


Figure 2

You may then find that you still have boxes where the contents are far from simple. How exactly do you 'Type in your program'?

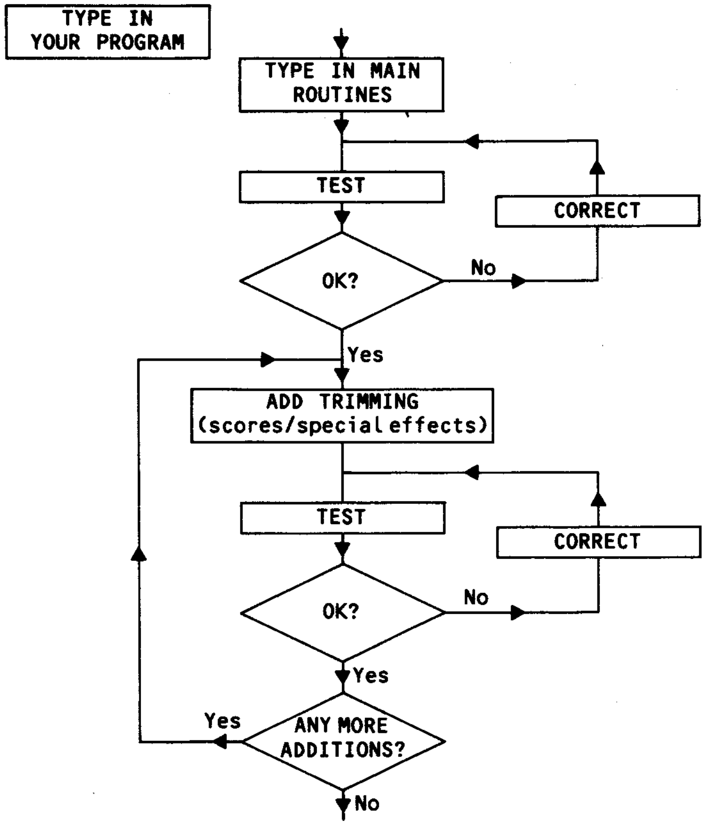


Figure 3

It doesn't finish there either, though the figures do! Clearly there is a lot more to 'Correct' than the one word, but you probably know your own de-bugging routines well enough not to have to bother writing them out.

Let these be your rules for flowcharting:

Always keep the overall structure of the program clearly in view.

Develop the details until you can see exactly what lines of BASIC you will need.

2
In the driving seat

You should already know how to produce the effect of movement by running HCHAR or VCHAR lines through a loop, so we can start from there.

```
10 CALL CLEAR
20 FOR C=1 TO 32
30 CALL HCHAR(10,C,42)
40 CALL SOUND(50,500,1)
50 CALL HCHAR(10,C,32)
60 NEXT C
70 GOTO 20
```

This simply runs an asterisk across the screen and makes an irritating noise. Now let's try and control that movement. We want to be able to change the Row number while the asterisk is moving. The only way to get information into the computer while it is running, without holding things up, is to use the CALL KEY routine. (Or the CALL JOYST routine, which in practice comes to much the same.)

We can add to our program so that the asterisk will move up a row whenever the E key (up arrow) is touched, and down when the X key is pressed. But first, our Row number must be a variable – so that it can be varied.

Add these lines:

```
15 R=10      (Row number at start)
55 GOSUB 100
```

and change

```
30 CALL HCHAR(R,C,42)
40 CALL HCHAR(R,C,32)
```

At line 100 we can then write the routine to collect information from the keyboard.

```
100 CALL KEY(3,K,S)
110 IF K=88 THEN 140      (CHR$(88) is X)
120 IF K=69 THEN 160      (CHR$(69) is E)
130 RETURN
140 R=R+1
150 RETURN
160 R=R-1
170 RETURN
```

Type it in and see how the program works now. You will notice that the program crashes if you try to fly off the top or bottom of the screen, but that is something that we can leave till later. Right now we will add some more controls – how about an accelerator and brake?

The speed of the program is largely controlled by the CALL SOUND line. If we make the time variable, we can change the speed of movement.

```
 6 T=50
40 CALL SOUND(T,500,1)
```

The A and B keys are here used as Accelerator and Brake, but you could use any other keys which you find more convenient. We need to add to our CALL KEY subroutine.

```
124 IF K=65 THEN 180      (65=A)
126 IF K=66 THEN 200      (66=B)
180 T=T-5      (speed up)
190 RETURN
200 T=T+5      (slow down)
210 RETURN
```

All typed in and running properly? Good. Now here's a way to get exactly the same effect, but with far less typing.

The value of truth (part 1)

Truth has a straight number value as far as the 99 is concerned. A statement that is true is worth −1. A false statement is worth 0. You can see this if you type in (no line numbers needed):

```
X=99
PRINT (X=99)
```

The 99 looks at the equation in the brackets and checks to see if it is true. It is, and so the 99 prints −1. Now type in:

```
PRINT (X=199)
```

This time 0 is printed.

We can adapt this to check the value of K from the CALL KEY line. Knock out line 110 and replace it with this:

```
110 R=R-(K=88)
```

Notice here that you have got a double negative. Take away minus one (− −1) is the same as 'add one'.

A similar line goes in for the E key.

```
120 R=R+(K=69)
```

Here you want 1 to be taken away when E is pressed, so you add minus one. + −1 is the same as −1.

Try it and see what happens. Watch those pluses and minuses carefully. Remember you have to stand on your head when you are valuing truth.

Everything OK? You are no longer using lines 140 to 170 so these can be knocked out as well.

We can take this one stage further, and save even more typing. You can include as many 'value of truth' functions as you like in one line. This means that lines 110 and 120 can be run into one:

```
110 R=R-(K88)+(K=69)
```

If neither key has been pressed both the brackets give 0 values and R remains the same. If one is pressed, you get the appropriate movement up or down. If both keys are pressed you get upward movement! Whenever the 99 find two or more keys down at a CALL KEY line it tends to pick out the one with the lowest character code. 'Tends to' – there are exceptions, and they don't follow any obvious rule. When you are using CALL KEY lines it is always worth checking out which keys have priority over others.

If you wanted to use 'value of truth' lines on the speed controls, where you are adding or taking away 5 each time, and not just 1, then you are going to need rather more complicated lines. We will return to them later. Meanwhile you might like to improve that first program by adding a nice graphic character to replace the asterisk.

```
5  CALL  CHAR(128,"00003098FEFF1830")
```

produces a little plane. Don't forget to change the code in line 30.

Sketchpad

You will have noticed in the earlier program the line:

```
CALL  HCHAR(R,C,32)
```

which printed a space over where the asterisk had been, so that you got a flickering movement. If you miss this out, you can develop a program to draw on the screen. This produces thick black lines:

```
10  CALL  CLEAR
20  CALL  CHAR(128,"FFFFFFFFFFFFFFFF")     (solid
30  R=5                                      block)
40  C=5       } (start point)
50  CALL  HCHAR(R,C,128)
60  CALL  KEY(3,K,S)
70  R=R-(K=88)+(K=69)
80  C=C-(K=68)+(K=83)  .
90  GOTO  50
```

Run this and try some computerised doodling. You might produce something like figure 4. (It can be done!)

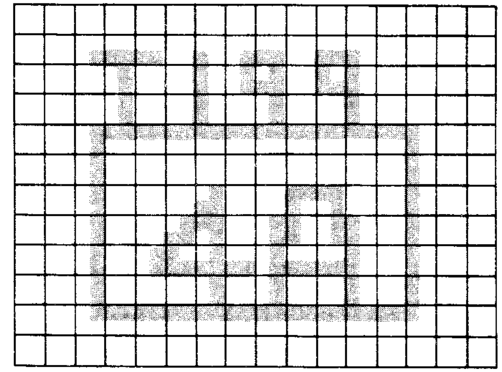


Figure 4

There's room for improvement, isn't there? The first thing to put right is the crashing when you wander off the screen. We will add a routine to fix that. Change 80 and add these lines:

```
 80  R=R-(R=0)+(R=25)
 90  C=C-(C=0)+(C=33)
100  GOTO  50
```

Lines 80 and 90 keep the Row and Column numbers within the limits of the screen. Whenever a number threatens to take the HCHAR position off the edge, then 1 is added or taken away to readjust it. We will come back to this again in the section 'The edges of the world'.

The second improvement is to give yourself some means of wiping out mistakes, and of moving from one part of the screen to another, without leaving a trail. We can do all of this with the same alteration, where we allow either a block or a space to be printed. The simplest way to do this is to make the printed character code into a variable. (G for Graphic). Line 50 now reads:

```
CALL  HCHAR(R,C,G)
```

Set the initial value of G somewhere earlier in the program.

```
35  G=128
```

We now make one of the keys into a switch, and look out for it after the CALL KEY line:

```
65  IF K=65 THEN 110      (65 = 'A', use another key
                          if you prefer)
```

This takes us to a routine to switch from block to space, or from space to block, for when you want to switch back.

```
110  IF G=128 THEN 140      (it is a block?)
120  G=128      (G must be 32 – the space)
130  GOTO 50      (and back to the print line)
140  G=32
150  GOTO 50
```

Type in the improvements and see how it works now. You should be able to draw a new range of doodles.

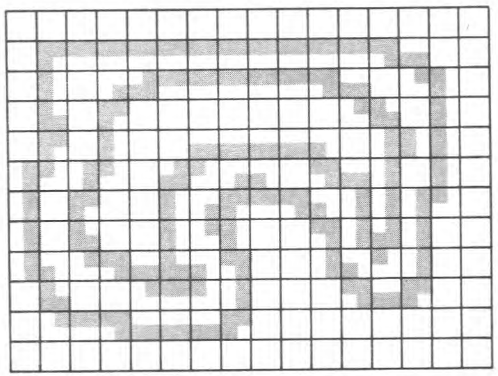


Figure 5

All very interesting, you might be saying, but what has this to do with games programs? The answer is 'several things'. Firstly it should help you to develop your ideas about steering and key-based controls. Secondly, you could use this sort of program as part of a larger one, where its purpose is to let you draw up a new game board each time you set up the game. Thirdly, it leads directly to 'Catch the Grimble', which we will come to in a little while.

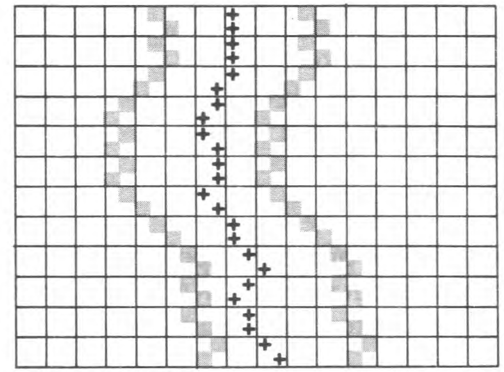


Figure 6

Meanwhile, here is the basis of a ski-run game which uses simple key controls. The game relies on the fact that the 99 starts printing from the bottom, and keeps scrolling upwards all the time. It prints the edges of a long and winding road, and also prints a 'skier' on that track. The player's job is to keep the skier inside the markers as they wind back and forth across the screen. This simply uses brackets for the edges of the track, and a plus sign for the skier. You may prefer to create some nice graphics instead and add them in at the beginning.

```
10  E=10      (Left-hand Edge column)
20  P=15      (Player's starting column)
30  PRINT TAB(E);"(";TAB(P);"+";TAB(E+10);")"
```

Note that the right hand Edge is always 10 spaces to the right.

```
40  X=RND
50  E=E-(X>.5)+(X<.5)
```

If the random number (X) is more than .5, then E will be increased by 1 and the track will move to the right. A small random number brings the track to the left.

```
 60  E=E+(E>20)-(E<1)      (keeps the track on
 70  CALL KEY(3,K,S)        screen)
 80  P=P-(K=68)+(K=83)
 90  IF  P<=E  THEN  120
100  IF  P>=E+10  THEN  120
110  GOTO  30
120  PRINT  "CRASH"
```

When you have got the program typed in and working, you might like to replace that simple 'CRASH' with a full routine. Some suitable sound effects and graphics and a few witty comments.

Let us look a little more closely at lines 90 and 100. You will see that there is a double check in each line. '<=' means 'is less than or equal to'. In this particular program, the equals sign alone would really have been enough, but there will be other times when you might just miss a 'collision' of this sort, and the double check makes sure that you don't. It takes very little space or time to include, and it might prevent some frustration. Make sure that the equals sign always comes second, or it may not work properly.

Those two lines could be combined into one if you prefer. You may remember from Starter Pack 2 that you can create AND/OR effects.

```
90  IF  (P<=E)+(P>=E+10)<>0  THEN  120
```

This single line does the job of the other two. If either of the equations in the brackets is true, then the total value of the two statements will be −1.

Game variations

1 The squeeze. Instead of having the right-hand side printed a fixed 10 spaces away, you could reduce the track width steadily. Start with a reasonable width:

```
5  W=10
```

Alter the print line so that the last part reads:

```
...TAB(E+ W);")"
```


and narrow the track before you return to the print line:

```
105  W=W-.1
```

This will reduce the track to nothing in one hundred lines, just over 4 screens full.

2 Speed-up. Here you build a delay into the program, but make the length of the delay variable.

```
  6  T=50
106  FOR  D=1  TO  T      (delay time)
107  NEXT  T
108  T=T-1
```

This has probably made rather a mess of your line numbering, so RESEQUENCE it to tidy it up again, SAVE it, and let the family play!

Joysticks!

If you have got them, you are probably itching to use them. If you haven't, go on to chapter three.

There is no doubt that the Wired Remote Controllers (to give them their proper name) make it much simpler to control movements on screen. You can actually feel the way you are trying to move your piece. They plug into the nine-pin socket on the left-hand side of the machine, and don't worry about plugging them in when you've got a program already loaded into the memory. The socket is protected so that your program is not disturbed.

MAKE SURE THE ALPHA LOCK IS UP whenever you are using joysticks. If you leave it pressed down the 99 will not pick up the forward movements properly.

The joysticks are linked into the program with a CALL JOYST line. This should state which joystick you are using, and give the variables where you want the movements to be stored. It is normal to use X for left–right movement, and Y for up and down. A line to read Joystick 1 would look like this:

```
CALL  JOYST(1,X,Y)
```

The numbers in the X and Y stores will always be either 0,4 or −4. There are 8 possible positions for the joystick, and the X, Y values of each are shown here.

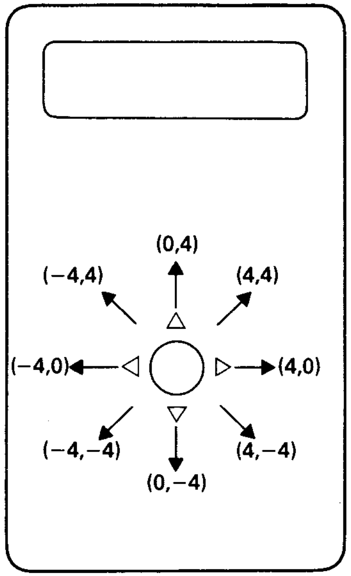


Figure 7

Let's build up a program to use the joysticks. This will move an asterisk around the screen. The asterisk's co-ordinates are stored in R and C, and these are adjusted by adding X and Y.

```
10  CALL CLEAR
20  R=12
30  C=16      (start in the centre)
40  CALL HCHAR(R,C,42)
50  CALL JOYST(1,X,Y)
60  R=R+Y     (vertical adjustment)
70  C=C+X     (horizontal)
80  GOTO 40
```

Type this in and run. Don't forget to check the ALPHA LOCK.

Not quite right is it? The asterisk is jumping 4 spaces at a time, and its working upside down. It is upside down because the Row numbers get bigger going down the screen, but the Joysticks numbers increase upwards. Change lines 60 and 70 to these:

```
60  R=R-Y/4
70  C=C+X/4
```

Now try it. See how close you can get to the edge of the screen without getting a 'BAD VALUE IN 40' report.

You might like to convert Sketchpad and Ski-run programs to work off joysticks.

There are, of course, two joysticks and you can, of course, use them both at the same time – or rather, you and another player can use them both at the same time. We will come back to them in the 'Two-player games' chapter.

3
Target practice

Shooting type games written in BASIC will never be as fast as machine code games, but speed is not the only thing that makes for a good game. Sound, interesting graphics and an element of chance all help to make a game more fun to play.

The program TARGET is a simple example of a shooting game, and this could be dramatically improved by the addition of some imaginative special effects and a good scoring system. There is nothing to stop you using TARGET as the basis of a game of your own. The flowchart for the program is shown in figure 8, and you will find it listed in Appendix A.

Shooting games don't have to be done this way, and it is worthwhile to look at the different routines that can be used.

Moving targets

A simple FOR. . .NEXT. . . loop moves the 'plane' across the screen:

```
350  FOR TC=1 TO 32        (Target Column)
360  CALL HCHAR(5,TC,128)     (128='plane'
...                            graphic)
650  CALL HCHAR(5,TC,32)    (rubbing-out space)
...
670  NEXT TC
680  GOTO 350
```

Notice how the graphic is printed at the start of the loop, but not rubbed out until very nearly at the end. This keeps the 'flicker' time down to the absolute minimum. In between these are fitted the various gun-moving, and hit-checking routines.

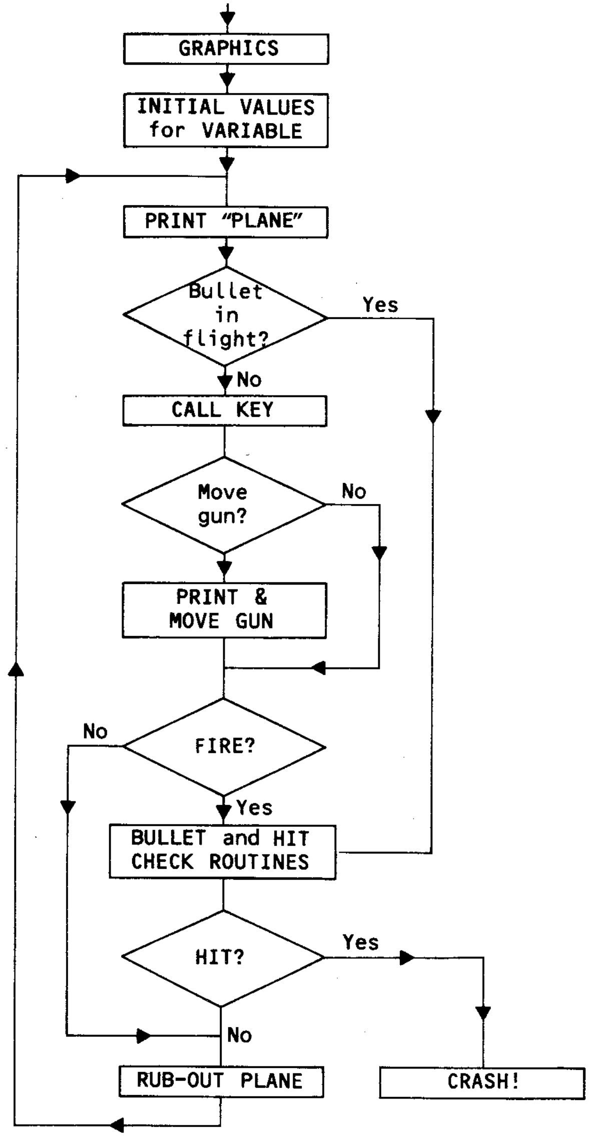


Figure 8

On this version, the plane always flies across at row 5. This could be made variable. It could be made to fly lower on each pass across the screen. This would give the player less time to respond. To do this you would replace the '5' in the CALL HCHAR lines with 'R', give an initial value to R, and add to it at the end of the loop.

```
345  R=5
675  R=R+1
```

Try adding these to the TARGET program and see what you think.

It actually makes it even harder than you think to hit the plane now. This is because the bullet skips 3 spaces at a time, so that it can pass the plane, but the hit isn't recorded. You can correct this by making line 675 read:

```
675  R=R+3
```

The crash routine will also need adjustment. It all goes to show that when you start fiddling with a program you always finish up with more work than you bargained for!

High speed bullets

In the present program you have a gun which can be shuffled across the bottom of the screen, and bullets which visibly fly up at the target. These could be replaced by a gun which could be steered anywhere about the screen. Then, when you have got the gun directly over the plane's position, pressing the Fire button will send an incredibly high-speed bullet zooming at the target. So fast, indeed, that it is invisible! Doesn't that make the program easier? The much simpler flowchart for this is shown in figure 9.
The 'Check for Hit line looks like this:

```
IF (TR=GR)*(TC=GC)=1 THEN...
```
(goto crash routine)

If it is true that both the row and the column co-ordinates of the target (TR,TC) and the gun (GR,GC) are the same, then you have $-1 * -1 = 1$.

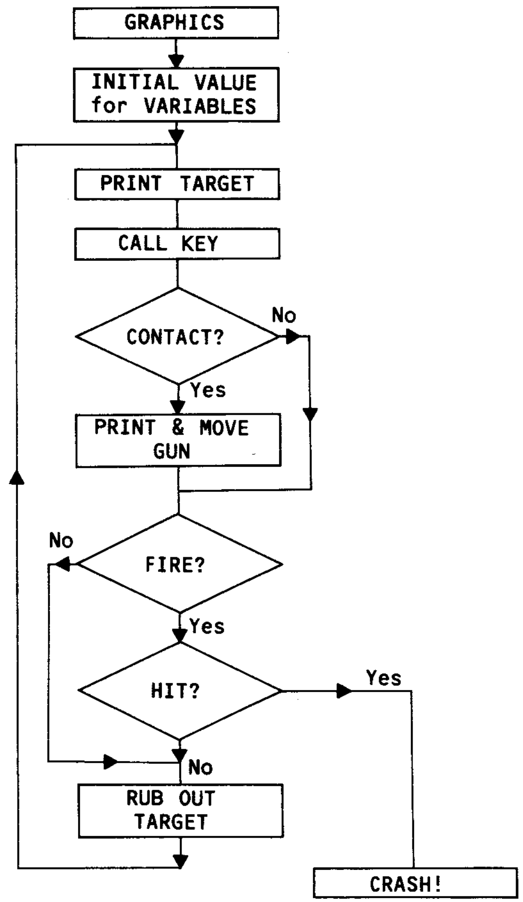


Figure 9

You might like to work out the BASIC program to produce that type of shooting game. A check program is given at the end of the chapter. Please remember that there is no single correct way of writing a program. If yours works, then that is all that really matters. Use the check program for reference only.

Checking for hits

Comparing co-ordinates is one way to check for hits, and works perfectly well, especially where you have only one target occupying only one space. If you had a larger target, or several, then the co-ordinate check lines would get rather complicated. Here is another way of checking. This uses the GCHAR subprogram. GCHAR is short for GET CHARACTER, and it will tell you what character is at a particular part of the screen. Try this:

```
10  CALL GCHAR(10,10,Z)
20  PRINT Z
```

Run it and it should print 32, the code for space. If it prints anything else then you must have had other material already printed on the screen. CALL CLEAR and run it again.

Now add this:

```
5  CALL HCHAR(10,10,42)      (or any other code
                              number you like)
```

This time it will print 42.

When you are using GCHAR check lines, you have to be careful to check the square before your bullet or gun is printed there, otherwise, you will simply find the code for that, and not for your target. In the TARGET program you will find these lines:

```
540  CALL GCHAR(BR,BC,Z)      (Bullet Row and
                               Column)
...
560  CALL HCHAR(BR,BC,129)     (129 = bullet)
570  CALL HCHAR(BR,BC,32)     (rub out immediately
                               for flickering effect)
580  IF Z=128 THEN 710      (128 = plane)
```

By waiting until the bullet has been printed and rubbed out before going off to the 'Crash' routine, you make sure that the target has been rubbed out as well.

Crumph!

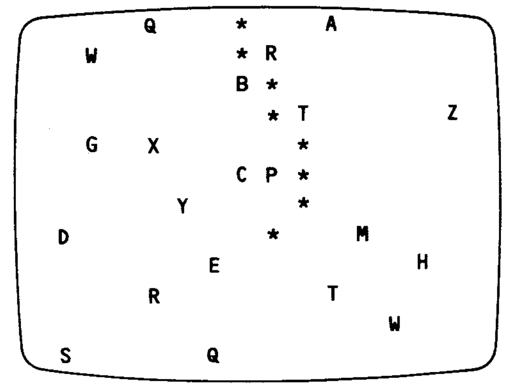


Figure 10

You can use the GCHAR check to find one particular character, or a range of characters. Look at the program below. This starts by printing random capital letters (line 50 works out the code number.) It then drops an asterisk down the screen. You, the player, have to steer the asterisk around the 'obstacles'. Notice the way that the check line picks up any character with a code over 64.

```
 10  CALL CLEAR
 20  RANDOMIZE      (don't forget this)
 30  FOR N=1 TO 24       (every row)
 40  X=INT(RND*32)+1       (random TAB position)
 50  A=INT(RND*26)+65       (random letter)
 60  PRINT TAB(X);CHR$(A)
 70  NEXT N
 80  C=15      (starting Column)
 90  FOR R=1 TO 24       (every row again, from the
                          top)
100  CALL GCHAR(R,C,Z)
110  IF Z>64 THEN 180       (hit something)
120  CALL HCHAR(R,C,42)
130  CALL KEY(3,K,S)
```

```
140  C=C-(K=68)+(K=83)      (left-right steering)
150  NEXT R
160  PRINT "MADE IT"      (you must have done to
170  GOTO 30              have got here)
180  PRINT  "CRASHED"
190  GOTO 30
```

Here's that check program for the 'high-speed bullet' game.

```
 10  CALL CLEAR
 20  CALL CHAR(128,"00003098FEFF1830")      (plane)
 30  TR=5      (Target Row)
 40  GR=15      (Gun Row)
 50  GC=15      (Gun Column)
 60  FOR TC=1 TO 32
 70  CALL HCHAR(TR,TC,128)      (print target)
 80  CALL KEY(3,K,S)
 90  IF S=0 THEN 150      (moving?)
100  CALL HCHAR(GR,GC,32)      (Rub out old gun
110  GR=GR-(K=88)+(K=69)       graphic)
120  GC=GC-(K=68)+(K=83)
130  IF K<>70 THEN 150      (firing?)
140  IF(TR=GR)*(TG=GC)=1 THEN 190
150  CALL HCHAR(GR,GC,43)      (prints a cross for the
                                gun)
160  CALL HCHAR(TR,TC,32)      (rub out old plane
170  NEXT TC                    graphic)
180  GOTO 60      (and fly across again)
190  FOR V=1 TO 30                          }
200  CALL SOUND(100,200,V,210,V,-8,V)       } (Bang!)
210  NEXT V                                 }
```

4
Two player games

Catch the Grimble

This is a steering game for two players. One player controls the Grimble, the other lays out Grimble cages. If the Grimble runs into a cage, or if the Grimble-catcher is able to drop a cage on it, then the game is over. In the version given below, there is no way in which the Grimble can stay free forever, but a simple counter keeps track of how long it stays on the loose.

The game produces screens something like figure 11.

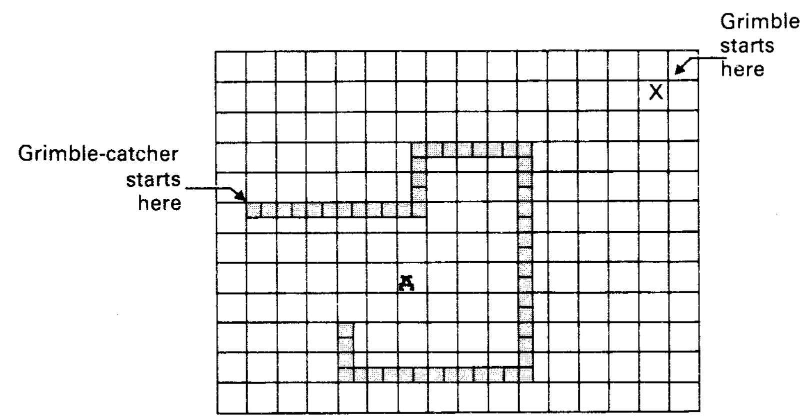


Figure 11

As there are two players, you will have to use the split-keyboard technique, or joysticks if you have them. The routines for the left and right sides can be combined into one, and we will return to that later, as it is probably easier at first to write in separate routines.

Here is the Grimble flowchart.

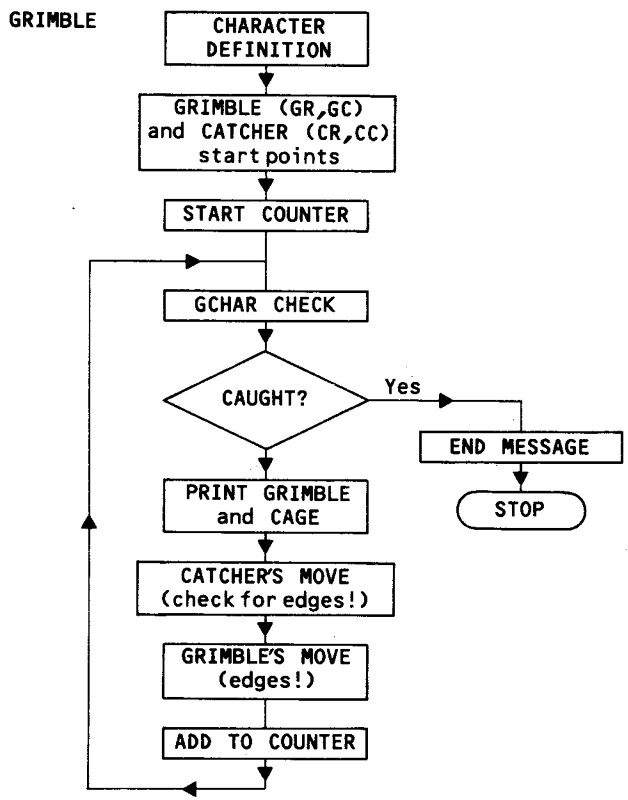


Figure 12

The split keyboard

The code numbers you get with CALL KEY(1. . .) and CALL KEY(2. . .) lines are quite different from the ASCII codes given by the standard keyboard check. The obvious choice for controls are the group of 'arrow' keys on the left hand side and the matching group on the right. Here they are with their codes.

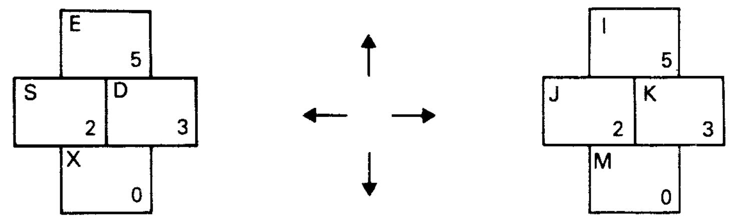


Figure 13

You would expect that the lines controlling up/down movement would look something like this:

```
R=R-(K=0)+(K=5)
```

Unfortunately, for reasons known best to itself, the 99 does not accept (K=0) as ever being true in this situation. There is always a solution though, and here is one.

```
...  CALL KEY(1,K,S)
...  K=K+1
...  R=R-(K=1)+(K=6)
```

You will have to add one to the column checks as well:

```
...  C=C-(K=4)+(K=3)
```

See if you can put 'Catch the Grimble' together, working from the flowchart. There is a check program at the end of the chapter if you need it. By the way, proper Grimbles look like this:

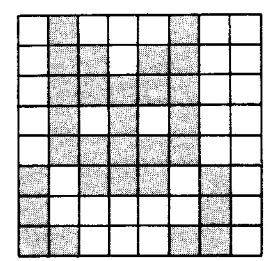


Figure 14

And this is a Grimble cage, unless you care to design a better one.

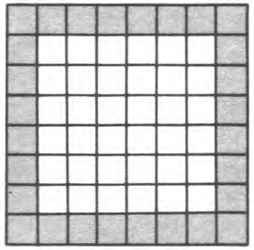


Figure 15

Game variations

1 Supper. It is little known fact, but Grimbles are very partial to your late night snack of cocoa and biscuits. Print a mug of cocoa on the screen, and scatter a few biscuits around. The object of the game now is to see how much of your supper the Grimble can scoff before it gets caught.

"FCFFFDFFFCFC78" gives this:

and

"3C429185A189423C" makes a Garibaldi:

Figure 16

2 Home. Draw a Grimble-hole somewhere along the bottom of the screen. Make its position random to give the Grimble a fair chance. It is now possible for the Grimble to win. You will need to include another check line to pick up when the Grimble reaches its hole, and an alternative ending to suit the occasion.

Grimble holes are quite large, and have specially shaped doors so that they can walk in without bending their feelers.

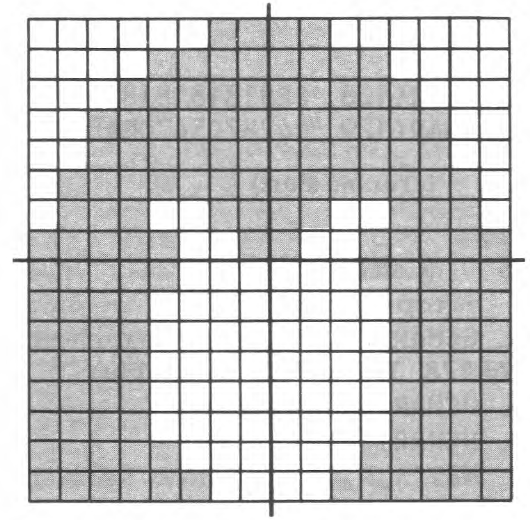


Figure 17

3 Compute-a-Grimble. You can get the 99 to look after the Grimble for you, but that requires quite a different approach. See 'Movement and Meetings in Mazes' below.

'Catch the Grimble' check program:

```
 10  CALL CLEAR
 20  CALL CHAR(128,"FF818181818181FF")     (cage)
 30  CALL CHAR(129,"44287C547CBA82C6")     (grimble)
 40  GR=1
 50  GC=32    (grimble start)
 60  CR=15
 70  CC=3     (Catcher start)
 80  T=0     (trip counter)
 90  CALL GCHAR(GR,GC,Z)     (cage check)
100  IF Z=128 THEN 280     (caught)
110  CALL HCHAR(GR,GC,129)
120  CALL HCHAR(CR,CC,128)
130  CALL KEY(1,K,S)     (catcher's movement)
140  K=K+1
150  CR=CR-(K=1)+(K=6)
160  CC=CC-(K=4)+(K=3)
170  CR=CR-(CR<=1)+(CR>=24)     (edge checker)
180  CC=CC-(CC<=1)+(CC>=32)
190  CALL HCHAR(GR,GC,32)     (rub out old Grimble)
200  CALL KEY(2,K,S)     (Grimble's movement)
210  K=K+1
220  GR=GR-(K=1)+(K=6)
230  GC=GC-(K=4)+(K=3)
240  GR=GR-(GR<=1)+(GR>=24)     (edge check again)
250  GC=GC-(GC<=1)+(GC>=32)
260  T=T+1     (trip counter)
270  GOTO 90
280  PRINT "YOU HAVE CAUGHT THE GRIMBLE"
290  PRINT "HE WAS FREE FOR";T;"TRIPS."
```

If you are using joysticks, the program is basically the same. Remove lines 130 to 160 and replace with these:

```
130  CALL JOYST(1,X,Y)
140  CR=CR-Y/4     (remember the joystick works the
150  CC=CC+X/4     opposite way to the Row numbers)
```

Remove lines 200 to 230 and replace them in the same way.

The dense pack theory of programming

If you look at the listing of DUEL you will find that one single routine is made to serve both tanks. In theory this is supposed to cut down on your typing time, and to produce a more compact and elegant program. In practice the program is indeed more compact, but the typing time is no less. The lines are quite complex, and mistakes are all too easy to make.

What happens here is that you use array variables rather than simple ones. R(1) stores the Row number for tank 1; R(2) for tank 2. Likewise C(1) and C(2) replace TANK1COL and TANK2COL (or whatever you would have called them).

When you come to arrange the lines for movement controls, you use a loop.

```
FOR P=1 TO 2
CALL KEY(P,K,S)
...
```

so that the first time it works as CALL KEY(1. . ., and next time round it checks the other side of the keyboard. (The CALL JOYST routine is handled exactly the same.)

The change of variables then looks like this:

```
R(P)=R(P)-(K=1)+(K=6)
C(P)=C(P)-(K=4)+(K=3)
```

and the check lines finish up with rather a lot of brackets!

```
R(P)=R(P)-(R(P)<=1)+(R(P)>=24)
C(P)=C(P)-(C(P)<=1)+(C(P)>=32)
```

Try converting the Grimble program to use arrays in this way. It may seem like a lot of work for very little reward, but there will be other times in your games writing where array use will save a lot of time, so practice now.

5
Changing directions

You might want a gun that can be pointed in different directions, or a target that spins when it is hit. You might want to manoeuvre a spaceship through the endless shoals of space. They all use much the same technique.

The first thing you need is a set of graphics that show the same object pointing different ways. The ones in figure 18 are from the RACETRACK program.

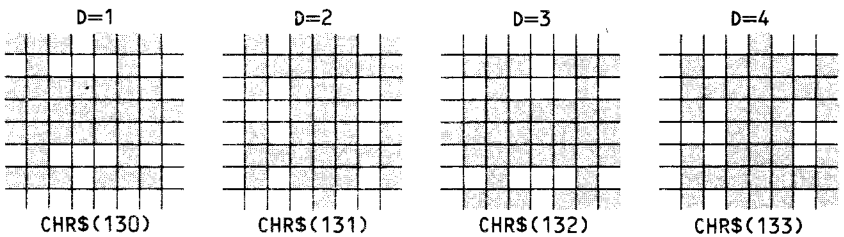


Figure 18

These are defined early in the program into character codes 130,131,132,133. This means that the graphic for any direction will be found at code 129+D.

When the car crashes into a wall, it is spun using a set of lines like this:

```
FOR D=1 TO 4
CALL CHAR(CR,CC,129+D)        (Car Row, Car Column)
NEXT D
```

Controls

These have to be rather different from the simple steering controls that we used earlier, as the 'car' is always moving forward – whichever way it is pointing. What is needed is an accelerator, a brake and some means of turning clockwise (right) and anti-clockwise (left).

As always, there are several possible solutions. Joysticks provide very simple controls for the player, and we will return to these later. If you are using Keys, then you might simply use the number keys 1 to 4 to fix direction, and letters A and B for speed controls. The routine would then look something like this:

```
CALL KEY(3,K,S)
IF K>52 THEN     (goto speed changing routine)
D=K-48
GOTO...
```

The line D=K−48 brings the code of the number down to its value. Code '1' is 49. 49−48 = 1.

This is not the method that you will find on RACETRACK. It may be simple to write the program this way, but the controls could prove confusing. There only two keys are used for steering. S (left, or anti-clockwise) and D (right). A quarter turn to the right is the same as D=D+1. A quarter turn anti-clockwise is D=D−1.

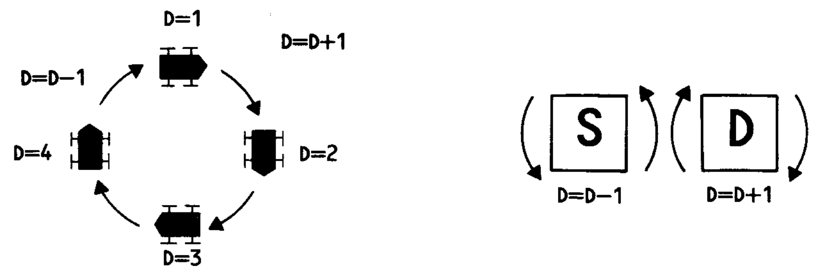


Figure 19

The routine then looks like this:

```
D=D-(K=68)+(K=83)
...
CALL HCHAR(CR,CC,129+D)
```

You need to slip a check line in there to stop D wandering out of range:

```
D=D-(D=0)+(D=5)
```

So if D = 0 it is increased to D = 1, and D = 5 is taken back to 4. This is a little crude. We will return to a better check in a moment.

Speed

How fast the car moves depends on the time value in the CALL SOUND line. This is variable, and in RACETRACK it is stored in M (speed of Movement). The keys E and X serve as accelerator and brake, and they could be made to alter the speed by a routine like this:

```
200  CALL KEY(3,K,S)
210  IF K=69 THEN 250
220  IF K=88 THEN 270
...     (direction changing lines)
250  M=M-10     (accelerator, reduces delay time)
260  GOTO...     (CALL HCHAR lines)
270  M=M+10     (brake)
280  GOTO...
```

However, if you look at the RACETRACK listing in Appendix A, you will find no such routine. Instead, you will find a variation on the 'value of truth' type of line. While this is somewhat harder to grasp, once you have got the hang of it, you will find that you save typing time, and get a slight increase in the speed of the program.

Time for a quick Detour.

The value of truth (part 2)

You know that a true equation is worth −1, but you can increase, or reduce, the amount of change produced by a true equation by multiplying it. Try this:

```
10  X=99
20  PRINT  10*(X=99)
```

Run it, and you will get −10. Alter line 20 so that X = something else and you will get 0. Put that back to X=99, and change the multiplier in line 20 to .5, and you will get −.5 as the result. The number you get at the end can be made positive by the use of a minus sign, and a set of brackets:

```
20  PRINT  -(10*(X=99))     (don't forget double
                             brackets at the end)
```

In RACETRACK this technique is used to produce a single line which alters the speed if either E or X is pressed.

```
...  M=M-(10*(K=88))+(10*(K=69))
```

A similar line prevents the CALL SOUND time from reaching 0, which would cause a program crash.

```
...  M=M-(10*(M=0))
```

Change that direction check line to:

```
...  D=D-(4*(D=0))+(4*(D=5))
```

and you will have smooth movement whichever way you steer.

Keep on moving

It is an important part of this sort of program that the car keeps moving, but you clearly cannot do this through a FOR. . .NEXT. . . loop, as you don't know where the car is supposed to be next. That is up to whoever is steering it. The change in the car's co-ordinates depends entirely on its direction at the time. You can see these changes in this table:

Figure 20

Direction	Movement	
D = 1	CC=CC+1	(to the right)
D = 2	CR=CR+1	(downwards)
D = 3	CC=CC−1	(left)
D = 4	CR=CR−1	(upwards)

By far the neatest way to change the car's variables is to use 'value of truth' lines.

```
CC=CC+(D=3)-(D=1)      (remember truth turns
CR=CR+(D=4)-(D=2)      everything upside down)
```

The alternative is a routine like this:

```
...  ON D GOSUB 1000,1020,1030,1040
...
1000  CC=CC+1
1010  RETURN
1020  CR=CR+1
1030  RETURN
...    etc.
```

ON. . .GOSUB works perfectly well here, where D must always be either 1,2,3 or 4, and the variable changes are very easy to see in those subroutines.

Turn and fire

If you want to develop a game like DUEL, where the tanks can fire in any direction, then the bullets' movement must be directed in the same way as the tank. Remember though, that you would normally want the bullets to travel faster than the tanks (or spaceships, guns, fire-breathing dragons or whatever). You can manage this in either of two ways. The bullet's movement could be run through a loop:

```
FOR T=1 TO 6      (or however many spaces)
BR=BR+(D=3)-(D=1)      (Bullet Row)
BC=BC+(D=4)-(D=2)      (Bullet Column)
CALL HCHAR(BR,BC,134)      (where 134 is the bullet)
CALL HCHAR(BR,BC,32)
NEXT T
```

You will need to fit a check line in there to spot any 'hits'. This gives a continuous movement and allows the victim no chance of escape.

The alternative is to use a variation of the 'value of truth' lines, as with the speed controls earlier.


```
BR = BR+(6*(D=3))-(6*(D=1))
BC = BC+(6*(D=4))-(6*(D=2))
```

This makes the bullet bound across the screen. You could splice this kind of bullet movement in with the main program, as with TARGET, so that your opponent has time to move. The bullet would then remain in motion until it hits its target or the edge of the screen. If you make the program jump over the direction changing routines when the bullet is in flight, then it will fly straight. Allow the program to run through the direction changer and you have a steerable bullet – a guided missile, no less!

Directional movement

What works for four directions works just as well for eight, but it's more than twice as much bother.

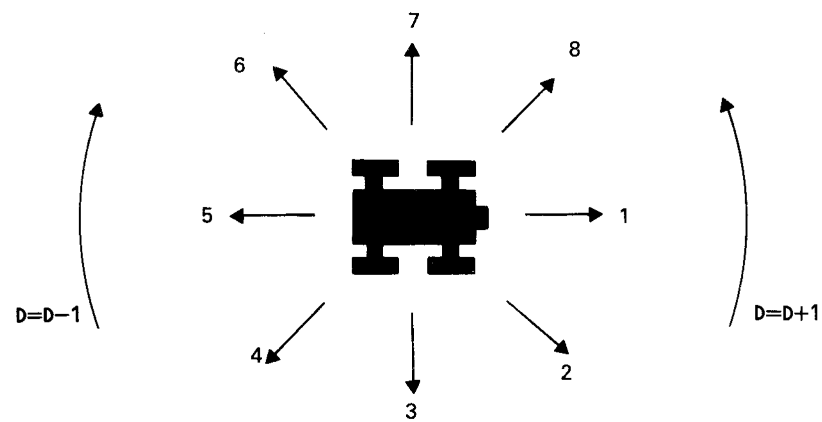


Figure 21

You will need eight graphics of course, and it will be more difficult to keep the same shape, as the new graphics will all be diagonal. It will be best to have a very simple shape with a clearly marked front end – a sharp point, or a long gun.

The turning routine can be exactly the same, except that you will need to change the upper limit in the check line from 4 to 8.

The main extra work comes in the movement lines. It will be much easier if you an ON D GOSUB. . . line, and have a set of eight subroutines. Four of these will simply change one variable each. The other four will have to each change two variables to allow for diagonal movement. This one moves up and right.

```
1100  CR=CR-1
1110  CC=CC+1
1120  RETURN
```

It is possible to make the changes through 'value of truth' lines, but they get terribly complicated. However, it is an interesting exercise if you feel up to the challenge.

Joysticks

If you have got joysticks you should use them for this sort of game. The program is simpler to write, and the controls are easier to use. The routine looks like this:

```
CALL JOYST(1,X,Y)
M=M-2.5*Y    (speed)
D=D+X/4      (direction)
```

The point you must remember when using CALL JOYST is that the X and Y numbers will be either −4,0 or 4. The X number must be divided by 4 to give one step at a time direction control. The Y value will also need adjusting to give the acceleration or braking that you want. Here it is multiplied by 2.5, so that speed is changed in steps of 10. Because the joysticks allow diagonal pressures it is possible to get both X and Y results at the same time, so that you can turn and brake in one movement.

6 The edges of the world

The question is, 'Does your 99 think the world is flat, round, or rubber-edged?' – Why not keep it guessing? You must do something when the spaceship/tank/car/duck reaches the edge of the screen, but it doesn't have to be the same thing every time. Here are your three main alternatives.

The flat earth

In this type of edge routine, you declare the player out whenever his piece goes over the edge of the screen.

```
IF (R<1)+(R>24)=-1 THEN...
IF (C<1)+(C>32)=-1 THEN...
```

Either line will send the program off to an end routine with some suitably silly comment like 'You have fallen off the edge of the world and the monsters have eaten you up.'

It's not the friendliest way to deal with screen edges, but it keeps people on their toes.

Flat Earth (1)

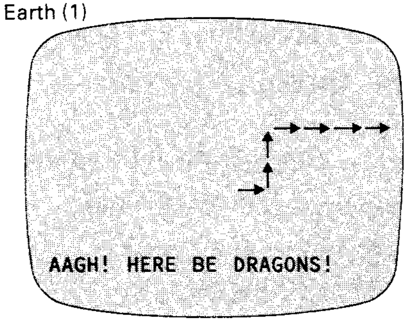


Figure 22

You have already been using another version of the flat earth approach, where there is a brick wall all around. Here the check lines prevent the variables from going beyond their limits.

```
R=R-(R<1)+(R>24)
C=C-(C<1)+(C>32)
```

You can, of course, use an actual 'brick wall' – well almost. Use HCHAR and VCHAR lines to draw a solid edge around your playing area, and use a GCHAR line to check the players' movements.

Flat Earth (2)

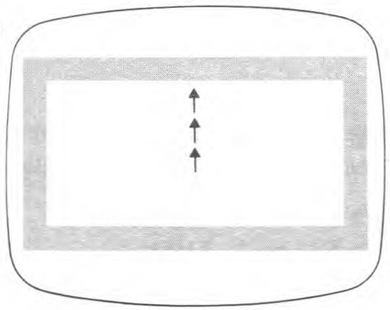


Figure 23

Wrap-around screens

These are for modern computers that know that the world is round. When a piece wanders off the edge, it reappears on the opposite side, as if it had shot round the back. You can do this with separate sets of lines for each edge:

```
... IF R>24 THEN...
... R=1
... GOTO...     (back to main program)
```

Or you can use two involved 'truth' lines:

```
R=R-(24*(R=0))+(24*(R=25))
C=C-(32*(C=0))+(32*(C=33))
```

This keeps the pieces in continual movement, and is especially useful if you are working out some kind of spaceship docking program. The ship could be steadily matched in speed and position with the 'space station', getting closer at each pass across the screen.

The wrap-around screen

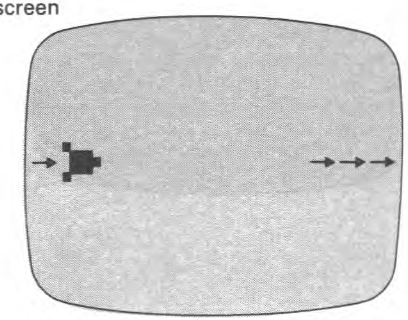


Figure 24

Rubber edges

Here the piece is bounced off the edge by altering its Direction control variable. Pick it up as it reaches an edge:

```
IF (R=1)+(R=24)=-1 THEN...
IF (C=1)+(C=32)=-1 THEN...
```

and change direction . . .

```
D=D+2
D=D+(4*(D>4))
```

This is for the 4 direction movement of course, and those D changing lines work for any directions, as you can see in this table.

D	D+2	D>4?	D−4	Original Direction	New Direction
1	3	no	–	Right	Left
2	4	no	–	Down	Up
3	5	yes	1	Left	Right
4	6	yes	2	Up	Down

Figure 25

Rubber edges (1)

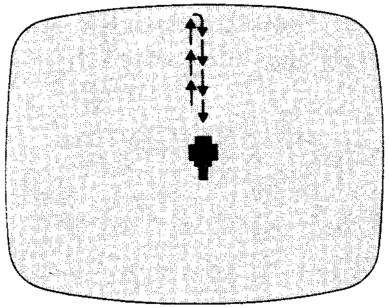


Figure 26

Diagonal bounces

These create difficulties all of their own. When you have only horizontal and vertical movement, the moving object will simply reverse direction on contact with the edge. However, when the movement is diagonal, the change of direction will be 90°. This would be no great bother, except that sometimes it will be 90° to the left, and sometimes 90° to the right. It all depends on the original direction, and the edge which has been hit.

You can see diagonal bounce routines at work in the BAT program. The 'bat' can only move diagonally, in the four ways shown below.

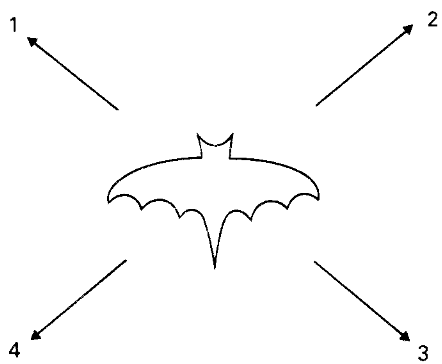


Figure 27

Here's what happens when he reaches the edges.

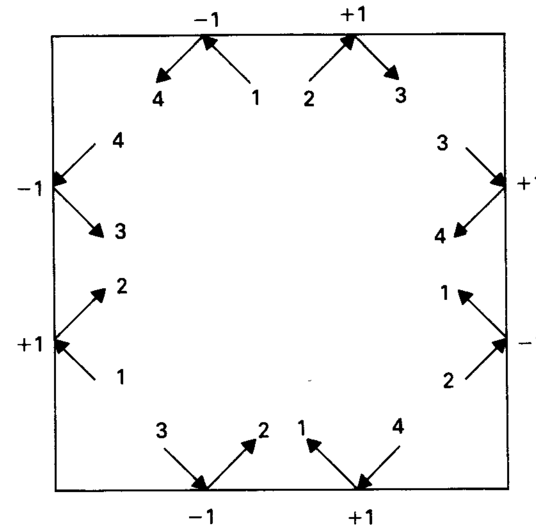


Figure 28

As you can see the direction change is not simple. The program must check the original direction, and the edge where the action is taking place. There are several possible solutions. The simplest, but longest is like this:

```
IF (D=1)*(R=1)=1 THEN...      (Direction 1 at top edge?)
...
D=4      (change to 4)
GOTO...      (back to main program)
```

You need 8 sets of lines like that.

Another method is used in BAT for the edge bounces. There the edges are coded. They may all look the same, but each edge uses a different graphics block with codes from 133 to 136.

A GCHAR line checks every square before the bat moves on to it. If the square has a code between 133 and 136, the program goes to the edge routine. (Lines 930 and on).

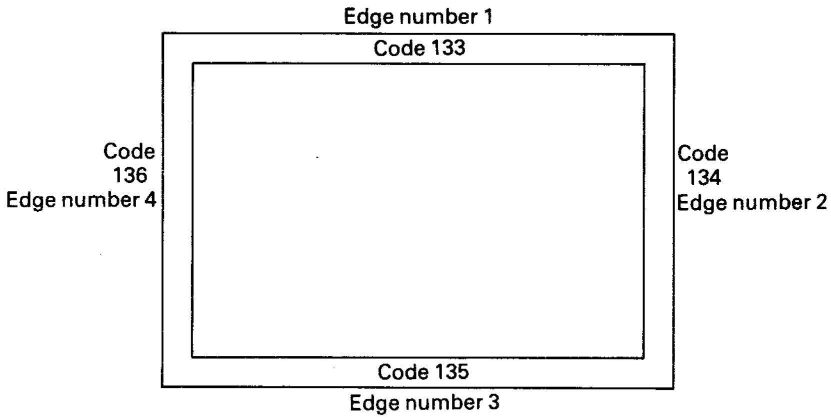


Figure 29

```
940  E=Z−132      (Z is the code got by GCHAR)
950  D = D + 1 −(2*(D=E))
960  D=D+(4*(D>4))
```

If you compare figures 28 and 29 you will see that when the direction (D) is the same as the edge number (E), then the change of direction is −1. If they are different the change is +1. It makes for simpler programming though to treat the −1 change as +3. It comes to the same thing in the end, and needs only a single check in line 960 to keep D in range.

Look what happens when the bat is flying up and left and hits the top. The original direction was 1, and the edge code is 1. Line 950 adds 1 and then adds a further 2 because the D and E variables are the same. The new direction code is 4. Contact with the left side changes this to 3. When the bat hits the bottom, coming from direction 3, his new direction code is 6, which is brought back to 2 by line 960.

The 'bat-knocker' works rather differently. It is assumed to have sides but no ends, so that the bat will continue in the same vertical direction, but with left and right swapped over. 1 becomes 2, 4 becomes 3, and vice versa. The change to D is therefore only ever 1 more or less, and it follows a simple rule. It is managed through this line:

```
D = D −(D=1)−(D=3)+(D=2)+(D=4)
```

1 is added if the original direction was 1 or 3, and taken away where it was 2 or 4. A double check line then keeps D within the 1 to 4 limits.

This type of routine can be combined with a straightforward reverse bounce routine to cope with 8-directional movement. When the missile hits the edge the program must work out whether a simple reverse or a diagonal bounce is needed. If you code your directions like this:

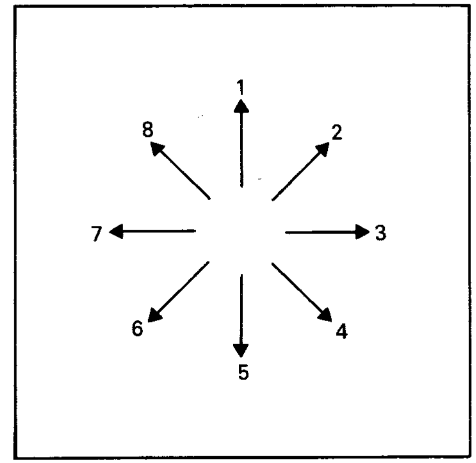


Figure 30

then you can pick up the diagonal bounces by the fact that the direction code is an even number. This line filters out even numbers:

```
IF D/2 = INT(D/2) THEN...
```

An odd number will end in .5, and this would be chopped off by the INTEGER function, and the numbers would therefore not be equal. Define your edge blocks into character codes 132,134,136 and 138, and you can get your edge code by taking 130 off the number produced by the CALL GCHAR line.

Sketch out your screen before you start and draw on it all the possible bounces. Make up a table of those bounces, divided into the simple reverse, and the diagonal types, and you should be able to see the numbers that you will have to use to change directions.

7
An element of chance

When a game gets predictable, it gets boring. If you know what's going to happen next, there's not much point in playing on. This is where you need to introduce an element of chance. (There is, of course, always the chance that your program won't work as you expect, but let's hope not!)

Random factors in shooting games

There is nothing to stop you from moving your target at random. If the target is a plane, you would expect it to fly smoothly, but it could vary its height as it flew. Hold the target row in a variable, and vary it with a line like this:

```
TR=TR +1+(2*(RND>.5))
```

If the random number in that line is less than .5 then 1 is added to TR and the plane dips. With a higher random number a further 2 is taken away (remembering that a true equation is worth −1). The result is that 1 is taken from TR and the plane flies higher. You will need a check line to keep the plane on the screen.

If the target is a duck, wild animal or alien spacecraft, then it might reasonably move by random jumps across the screen. This routine produces jumps of between 0 and 3 columns:

```
J = INT(RND*4)
TC=TC + J
```

The target might fire back, or drop bombs, as happens in the standard Space Invaders game. You will then need to work in for the target the same kind of routines that you have for the gun. Is it firing or isn't it? This can be controlled by a line like this:


```
TF = (RND>.5)
```

The Target Fire variable is therefore either −1 or 0. Another line will send the program to a bomb routine if appropriate:

```
IF TF THEN ...
```

Note that IF TF. . . means the same as IF TF =−1, indeed it means IF TF is anything other than 0.

Bomb routines are the same as bullet routines, though going in the opposite direction! You will find that the program runs slower when you are asking the computer to handle a target, a gun, a bomb and a bullet all at the same time. This is inevitable in TI BASIC, but you can improve the speed of programs by working in EXTENDED BASIC, where SPRITES give you smoother movement at about twice the speed. (See Appendix B)

A hit doesn't have to be fatal. You might only damage the target – or it might only damage you. The amount of damage can be random.

```
TD=0                 (Target Damage at start)
...
TD = TD +(RND*10)    (how much damage this time?)
IF TD>20 THEN...     (off to 'shot down in flames' routine)
```

In this example the target would receive, on average, 5 points of damage, so you would expect to have to hit it at least 4 times to knock it out completely. The figures should be adjusted to suit how you want the game to run.

Guessing games

Playing a guessing game with the computer should be like playing with another person. You should not be able to predict the answer; you will want to know when you are right and sometimes you will expect to be given some clues as to how you are doing, when you get things wrong.

In Starter Pack 2 you will find a 'Hunt the Thimble' game. The object of that game was for the player to guess a pair of co-ordinates selected by the computer. 'Colder-warmer' clues are given to help the player find the hidden spot. To find out whether a guess is better or worse than the previous one, the 99 calculates the total difference between the thimble's co-ordinates and the guess. This was done by finding the absolute difference between the guessed and real row co-ordinates, and between the guessed and real column co-ordinates. The total of the two is the overall difference. Y and X are the 99's numbers, R and C are the player's.

```
D1 = ABS(Y-R)     (vertical difference)
D2 = ABS(X-C)     (horizontal difference)
D = D1 + D2
```

Because the ABS function knocks off the minus sign (if there is one), this routine always picks up the total difference, wherever the guess might be. You can see the effect of some guesses in figure 31.

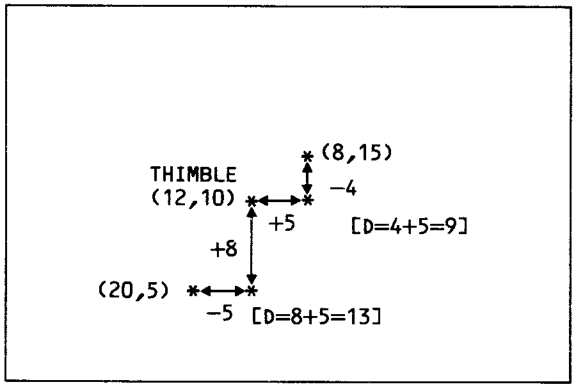


Figure 31

An alternative way to work out differences like this is to use Pythagorus' rule. There, if you ever wondered what the ancient Greeks could offer the modern computist, now you know!

'The square on the hypotenuse is equal to the sum of the squares on the two other sides.'

The distance between (Y,X) and (R,C) can be worked out like this:

```
A = Y-R
B = X-C
D = A^2 + B^2
```

(A^2 means A^2)

This can be packed into one line if you prefer:

```
D = ((Y-R)^2)+((X-C)^2)
```

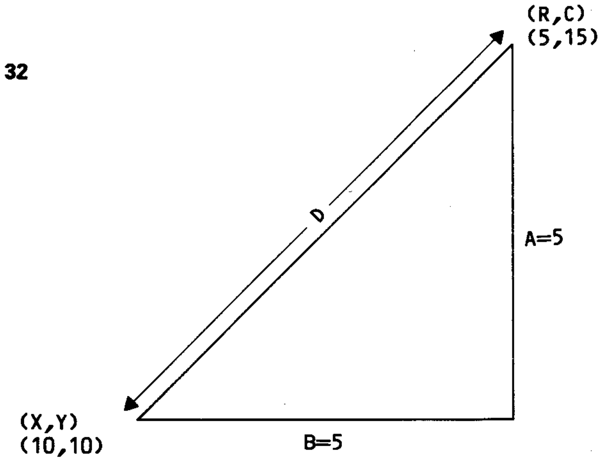


Figure 32

'Pick a straw'

A simpler type of guessing game – indeed, probably the simplest type – is the 'Pick a Straw' played by the gambling Goblins in DRAGON. In that one, whichever straw you choose, you have a 50/50 chance of being wrong. The flowchart for the routine is given in figure 33.

If you look at the program list for DRAGON you will find the gambling routine at lines 2000 onwards. This could be rewritten as a new gambling game using 'Heads or Tails' instead of Left or Right Straws. You would need some good graphics and a nice clear print out of the player's and the 99's cash balances. Why not start out with £1 million each and play a double or quits game, with no limit on the stakes.

For more complicated gambling games, have a look at the cards and dice games in Games Pack 2.

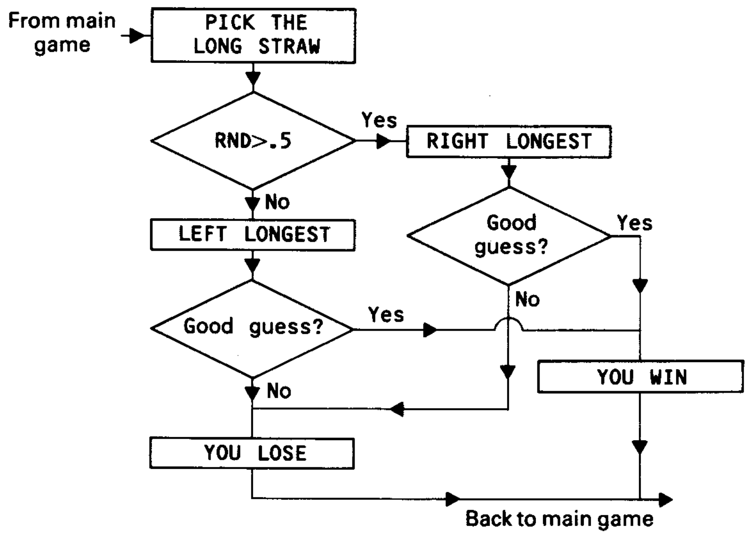


Figure 33

8 Obstacles and random dangers

In Ski-run and Crumph games given earlier the player could see the obstacles that had been put in his path. These obstacles do not need to be visible. They are hidden in the next program, 'Minefield', by colouring them transparent.

```
 10 CALL CHAR(128,"FFFFFFFFFFFFFFFF")
                                          (a block)
 20 CALL COLOR(13,1,1)       (but 'see-through')
 30 CALL CLEAR
 40 FOR N=1 TO 50
 50 X= INT(RND*24)+1
 60 Y= INT(RND*32)+1         (this scatters 50 mines)
 70 CALL HCHAR(X,Y,128)
 80 NEXT N
 90 R=1
100 C=1          (player's start)
110 CALL HCHAR(R,C,42)
120 CALL KEY(3,K,S)
130 R=R-(K=88)+(K=69)
140 C=C-(K=68)+(K=83)
150 CALL GCHAR(R,C,Z)        (check the square before
                              moving)
160 IF Z=128 THEN 180         (trod on one)
170 GOTO 110
180 CALL COLOR(13,2,1)        (so you can see where
                               they are)
190 CALL SOUND(1000,-3,1)
```

You will need to add a 'home safe' point, and write in a check line for it, and the end of the program needs tidying. Hold the screen with a CALL KEY and then offer the player another go. If you find that the minefield is too dangerous for your taste, then reduce the number of mines by altering line 40.

The game could be made friendlier by equipping your player with a 'mine-detector'. This can be managed in two different ways.

The first way is to print 'warning squares' (also invisible) around each of the mines.

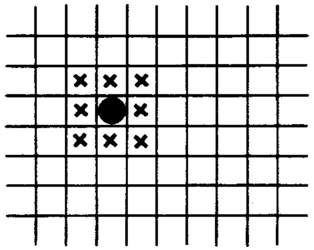


Figure 34

Here the mine is at 8, 9. The warning square routine looks like this:

```
FOR N= 1 TO 50
X= INT(RND*22)+1        (gives numbers from 1 to 22)
Y= INT(RND*30)+1        (between 1 and 30)
FOR T=0 TO 2
CALL HCHAR(X+T,Y,129,3)     (129 = warning
NEXT T                      square)
CALL HCHAR(X+1,Y+1,128)     (the mine)
NEXT N
```

You will see that this first prints the warning square blocks, and then adds the actual mine on top. The X and Y random limits had to be changed slightly to make sure that the warning areas stayed on the screen.

A further routine now needs to be added so that if code 129 is picked up by the GCHAR line, a warning beep sounds.

The second sort of 'mine detector' uses a looped GCHAR line to check all the squares around each move:

```
FOR N = -1 TO 1
FOR T = -1 TO 1
CALL GCHAR(R+N,C+T,Z)
IF Z = 129 THEN...      (warning sound)
NEXT T
NEXT N
CALL GCHAR(R,C,Z)
IF Z = 128 THEN...      (boom!)
```

Notice how the FOR. . .NEXT. . . loops check either side and up and down from the move square. That particular square needs to be rechecked later to see if it is a mine, as the looped check only gives warnings.

These Minefield programs use the screen itself to map the game. If the screen has to be cleared, or is altered by INPUT or PRINT lines, then the map is ruined, or lost altogether. This makes no difference here, as you would want to have a new layout each time you played. However, if you wanted to give your player several tries at each layout, you would run into difficulties. One solution is to store the map in an array. You will remember from Starter Pack 2 that an array is a set of stores, all with the same name, but with different reference numbers (or subscripts). These numbers can start from 0 or from 1. Throughout this book it is assumed that you will write OPTION BASE 1 in your programs, and that the arrays will therefore start from 1.

The line DIM M(24,32) sets up a bank of stores that is 24 rows deep and 32 columns wide – the same size as the screen. When the stores are first opened they all have a value of 0. This can then be altered (at random) to code in your mines.

```
X= INT(RND*24)+1
Y= INT(RND*32)+1
M(X,Y) = 1
```

You do not need to transfer the map to the screen to check for hits. It is sufficient to check the array.

```
IF M(R,C)=1 THEN...
```

Set up a 24×32 array and write a loop to scatter 50 or so 'mines' through it. You can then get it printed out like this:

```
FOR R=1 TO 24
FOR C =1 TO 32
N=M(R,C)     (find the number at each point)
CALL HCHAR(R,C,48+N)
NEXT C
NEXT R
```

There is a catch to using simple number arrays like M(24,32) as game maps, and it is that they consume an enormous amount of memory. Each store within a number array takes 8 bytes – this is so that very large, or very small numbers could be stored there if wanted. This means that M(24,32) takes a total of 6144 bytes. Actually it takes 6154, as a further 10 bytes are needed to organise the array. A string array, on the other hand, is much more economical in its use of memory. Each string store takes up only 2 bytes, so M$(24,32) takes a total of 1546 ($24 \times 32 \times 2 + 10$).

A string array is used in the DRAGON program, both to map out the path (see below 'Mazes') and also to scatter the goblins, gold and dragons through the maze. The routine which does this goes from line 530 down. If you wanted to have a look at the array before you play the game – purely for research purposes, and not so that you can cheat – then add:

```
 655 GOSUB 7000
7000 FOR R = 1 TO 21
7010 FOR C = 1 TO 21     (the array (P$) is 21 × 21)
7020 IF P$(R,C)="" THEN 7050     (string arrays
7030 PRINT P$(R,C);              are empty at
7040 GOTO 7060                   the start)
7050 PRINT " ";     (a space to fill any gaps)
7060 NEXT C
7070 PRINT     (moves print position to next line)
7080 NEXT R
7090 INPUT A     (a wait-a-bit line)
7100 RETURN
```


You should see something not unlike figure 35. '1' indicates path, '2' is a crock of gold, '3' a dragon and '4' a goblin.

```
1 1 1 1 1 1 1 1 1 1 1
2 1 1 1     1       1
  1                 1 1 1 1 3 2 1 4 1
  1                   1 1 1         1
  1               1 2   1 1 1       1
  1 1 1             1   4 1 1       1
      1             1 1 1 1 3 1 1 1 1
      4 1 1           1 1 4 2 1 1
      1                   1 1 1 1 1 1
      1                   1 1 1 1 1 1 1
                          2 1 1 1 3 1 1
    1                     1 1 1 1 1 1 1
    1   1           1 1 1 3 1 1 1   1
  1 1   1               1   1   1   1
    1 1 1               1 1 4   1 1 1
        1                   1   1   1
        1           1       1   1 1 1
        1   4   1   1           1   1
        1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Figure 35

All this should have whetted our appetite for mazes, which is just as well, because here they come.

9 Mazes

There are two types of maze. The first has a fixed path and is usually a field on which a shooting or chasing game is played. 'Munchman' is a classic example of this sort of game. A maze of this type is really a complicated obstacle course, and is designed in the same way.

The second type of game has a disguised path, and the object of the game is to find the way out. The game can be made more interesting by including a number of incidents for the player to meet and deal with on the way. DRAGON is an example of this type. You will notice that not only is the path hidden, it is also different every time you play. The dragons and goblins are also randomly positioned as mentioned in the last chapter.

Random paths

A random path is one produced by a series of random moves, up, down, left or right. This routine shows a simple random move routine:

```
 10  CALL CLEAR
 20  R=12
 30  C=16        (start in the middle)
 40  CALL HCHAR(R,C,42)
 50  X= INT(RND*4)+1     (1,2,3 or 4 at random)
 60  ON X GOTO 70,90,110,130
 70  R=R+1
 80  GOTO 40
 90  R=R-1
100  GOTO 40
110  C=C+1
120  GOTO 40
130  C=C-1
140  GOTO 40
```

Type this in and watch the asterisk wander about the screen. As there is an equal chance of it moving in any direction you will find it tends to produce a wadge in the middle of the screen, like figure 36.

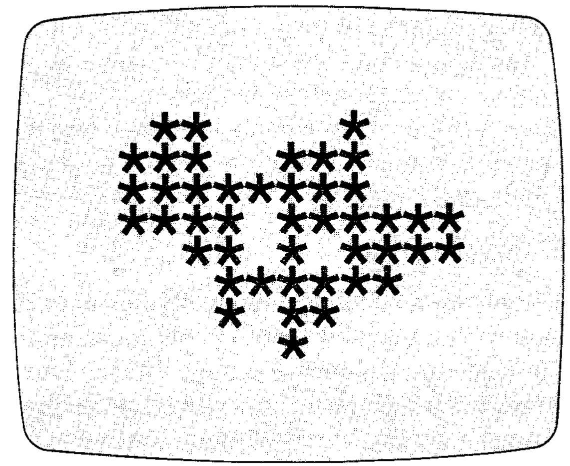


Figure 36

You need a better a method of sorting out those random numbers if you want to produce a path that actually goes somewhere. The MAZE program uses a routine like this:

```
620  X= RND
630  IF X>.85 THEN...     (left routine)
640  IF X>.5 THEN...      (right routine)
650  IF X>.35 THEN...     (up a row)
660  down a row starts here
```

Line 620 fixes the random number for this trip round the step-making loop. The next three lines filter out the higher values of X and send them off to the left, right and up routines. Any number less than .35 produces a downward move. There is an even chance that the random number will lead to a vertical or a horizontal move, but there is then a bias built in to make the right and down moves more likely than the left and up ones. Run the MAZE program and you can watch the whole routine at work.

MAZE is programmed to find a path from 1,1 to 10,10 on its first run through. When it has reached the end, you can enter your own start and end co-ordinates.

The random limits in lines 630 and 650 are then altered to produce a suitable bias to the path.

Line 630 is actually written as

```
630 IF X>X2 THEN...
```

X2 has an initial value of .85. It will be changed to .65 or .75 if the positions of your start and end points mean that the path must head left, or remain on the same column. The program works best when the end point is on an edge. It can very easily overshoot a central 'end-point' and wander off across to the opposite side!

The hidden path

You can create a concealed path by printing transparent paving slabs on the screen, in the same way that the 'Minefield' program used transparent mines. A more flexible method is to use an array.

We can now put together the things covered so far to make the first part of an array-based maze program. Here's the flowchart.

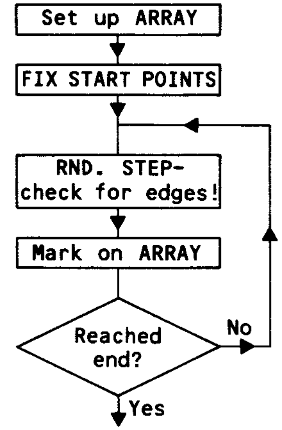


Figure 37

And the program looks like this:

```
10 OPTION BASE 1
20 DIM M$(10,10)
30 R=1
40 C=1
50 X= RND
60 IF X>.85 THEN 150
70 IF X>.5 THEN 130
80 IF X>.35 THEN 110
90 R=R+1+(R=10)
100 GOTO,160
110 R=R-1-(R=1)
120 GOTO 160
130 C=C+1+(C=10)
140 GOTO 160
150 C=C-1-(C=1)
160 M$(R,C) = "1"     (any character could be used)
170 IF (R=10)*(C=10) THEN 190
180 GOTO 50
190 ...     (next part of program)
```

In the program above the path is made up of '1's, but it could equally well be a defined graphic block. If you add:

```
5 CALL CHAR(128,"FFFFFFFFFFFFFFFF")
```

and alter line 160 to:

```
160 M$(R,C) = CHR$(128)
```

Then the print routine will produce a path of blocks.

It is probably worthwhile at this stage to add a routine to print up your maze, just so that you can see it works. We can adapt it for game purposes later. The one given below is basically the same as the one suggested at the end of the last chapter, but here we are using HCHAR lines to print on the screen.

```
190 CALL CLEAR
200 FOR R=1 TO 10
210 FOR C = 1 TO 10
220 IF M$(R,C) ="" THEN 250
230 N = ASC(M$(R,C))      (finds code of character in
240 CALL HCHAR(R,C,N)      array)
250 NEXT C
260 NEXT R
270 INPUT A    (to hold the screen)
```

This prints the path as it really is, but we could disguise its appearance by scattering 'imitation paving stones' about the screen. They would look like the real ones that make up the path, but they would not be present in the array.

The trick blocks can be laid by slipping these three lines in after 240:

```
242 GOTO 250       (so the routine is jumped after a
                    proper move)
244 IF RND>.5 THEN 250
246 CALL HCHAR(R,C,128)    (assuming 128 is your
                            path code.)
```

Now alter line 220 so that the program jumps to 244 when it reaches an empty store in the array.

Try the program out, at first without those extra random 'paving slabs' and then again with the random routine included. Alter the random limit in line 244 and see what difference it makes to the appearance of the path.

Another way to confuse the player is to have the 99 draw some misleading paths as well as the main one through the array. Ideally these extra paths should go from nowhere to nowhere, but cross the main path at some point. This is what happens in DRAGON.

Four trails are started from fixed points within the array, and each wanders off for a maximum of 20 steps before coming to a sudden stop. The effect can be quite confusing. As the path-making routine is used several times, it has been made into a sub-routine. The flowchart for the 'paths' section of the program is shown in figure 38.

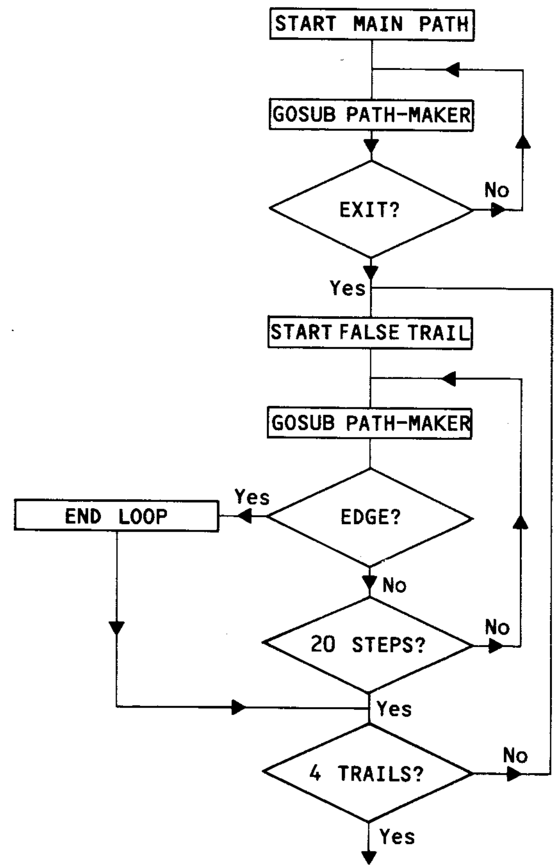


Figure 38

The main path routine starts at line 200 in the program.

```
200  DIM P$(21,21)
210  R=2
220  C=2
230  GOSUB 4000
240  IF (R=20)*(C=20) THEN 260
250  GOTO 230
```

You will notice that the array here is 21 squares each way. The path within is kept between 2 and 20. This leaves a 'wall' around the outside to stop the player escaping.

False trails

The routine for these starts at 260:

```
260  FOR T=1 TO 4
270  R=T*3            } (so the start points are scattered
280  C=16-R           } diagonally across the map)
290  FOR N=1 TO 20
300  GOSUB 4000
310  IF (R=20)+(C=20) THEN 330     } Maximum 20
320  GOTO 340                      } steps
330  N=20
340  NEXT N
350  NEXT T
```

That check line at 310 stops a path when it reaches the bottom, or the right hand side. Without it, there would be a danger of the false trail leading to the exit, and that would not do.

The full listing of DRAGON is given in Appendix A. You may like to look at that path-making subroutine. It is not quite what you would expect. The path is built two steps at a time. This stretches the paths out, and produces a better maze, but is more complicated than a single step routine.

The main problem is that when you mark off the path in the array, you need to mark the squares that have been jumped over, as well as the ones that are 'landed on'. Figure 39 shows this.

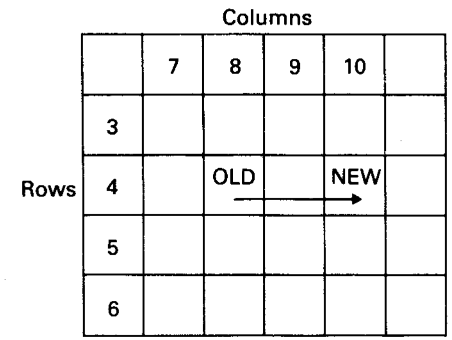


Figure 39

Each move now needs a set of lines like this:

```
4040  R=R+2+(2*(R>19))
4050  P$(R-1,C) ="1"
4060  GOTO 4150
....
....
4150  P$(R,C)="1"
4160  RETURN
```

You will see that the check in line 4040 is also more complicated.

10 Movement and meetings in mazes

When you have a maze handled by an array, it is not necessary to actually show the movement through it on screen, or indeed to show the maze at all. Many adventure games of the 'Dungeons and Dragons' sort simply tell you what you can see. It's up to you, the player, to work out where you are. These mazes are designed, usually in three dimensions, as a series of rooms linked by passages and stairways, with plenty of dead ends and sudden drops. At the simplest level the screen display is a set of print lines. These will tell you things like 'There is a passage on the right, and one on the left. In front of you is a door. It is closed. Do you want to (1) turn left, (2) turn right, (3) open the door?' This is followed by an INPUT A line.

Movement through the 'dungeon' in this kind of game is then controlled by the player's inputs:

```
ON A GOSUB...
```

The subroutines will alter the player's co-ordinates to suit the movement, and will deal with any meetings.

The appearance of this sort of game can be improved by including routines to give a 'view'. (Figure 40)

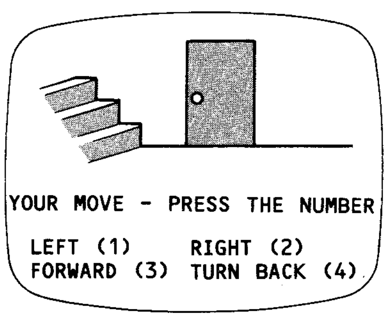


Figure 40

Two-dimensional mazes could also be treated this way, or mapped on to the screen as in the DRAGON program. There the 'hero' clears a path behind him as he works his way through. This makes it much easier to retrace his steps if he comes to a dead end. You don't have to do this. Your maze games might only show the piece on screen when it meets something. You might not even give your player that much. You could leave him groping blindly in the dark, trying to work out where he is by remembering each move. This cuts out a few bothersome screen routines, but is not particularly friendly of you. However, some people like their games hard. You could print up on screen where some, or all of the incidents are. They might be there from the beginning, or appear when the player has earned the extra information. (See Colour Changing)

Controlling movement

If you are displaying movement on the screen, then you will not want to have that movement controlled by INPUTTING left, right, up down instructions. The INPUTS will ruin the screen layout, unless you use the special Input Anywhere routines that were covered in Starter Pack 2. It is far better to use a simple CALL KEY line linked to the 'arrow' keys (ESDX), in the same way as in the shooting and steering programs. This must then be followed by a routine to check the square ahead to see if movement is possible, and if there is something at that square. Here's a flowchart for this part of a maze program. You might like to compare it with the lines from 840 onwards in the DRAGON list.

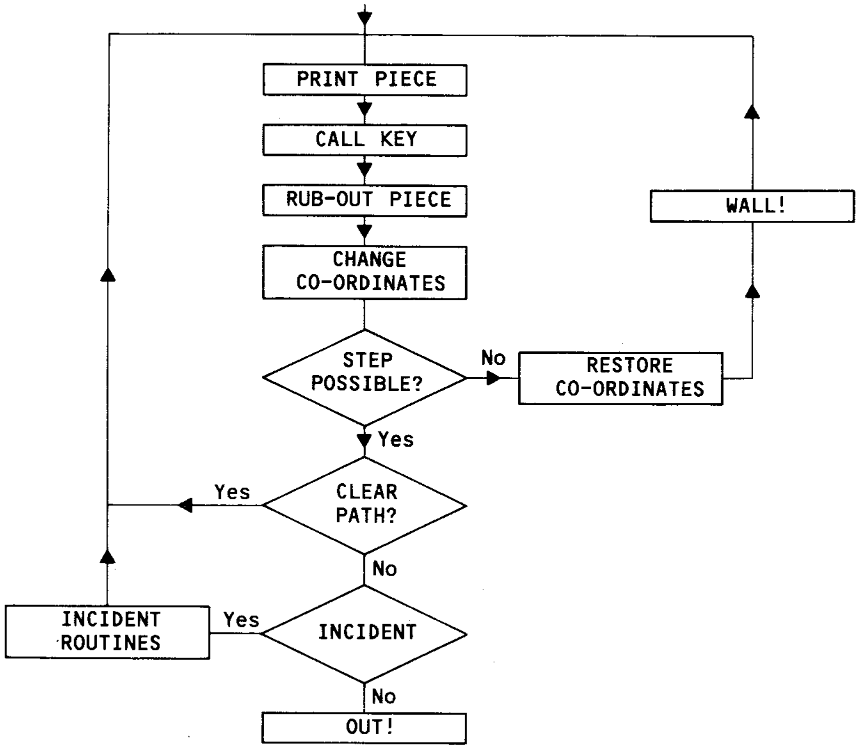


Figure 41

Meetings

You will normally want to include incidents of some sort in your maze, to make the game more interesting. 'Fight your way through hoards of evil glorks to rescue the beautiful princess and claim the sacred sword of Scaramonca' sounds much more fun that 'Find your way out'.

The routines, or subroutines, that handle the incidents can be as long as your imagination and the TI's memory will allow. As a rough and ready guide, the DRAGON program takes up about 8k of memory when it is running. There is room then for a maze program with a larger (three-dimensional) maze and more complicated incident routines, or a wider variety of incidents. Take care that your program does not take up more than 12.5k, or you will not be able to save it properly. This 12.5k does not include the space taken by arrays and other variables when the program is running. The DRAGON program alone takes just over 6k, with the extra 2k needed as workspace.

Fixed incidents

Bags of gold, traps, stationary dragons or sleeping monsters – these are scattered through the array using a routine similar to the one covered in 'Obstacles and random dangers'. The only difference is that the routine has been extended to scatter a random variety of incidents. Look at line 530 to 650 in the DRAGON list.

Moving dangers

Your dragons and monsters do not have to stay still and wait for the hero to find them. They could move through the maze looking for him! To manage this you will need to combine the techniques of movement used in the targets programs with the path-drawing routines used in your maze.

Start by indicating his presence with a variable. 1 for alive, 0 for dead.

```
M=1
```

Give him a start position early on in the program, making sure that he is on the path:

```
... MR =INT(RND*18)+3
... MC =INT(RND*18)+3
... IF P$(MR,MC) ="" THEN ...      (back and try again)
... P$(MR,MC) = "6"     (where "6" is the monster code)
```

At some point in the main game loop, you send the program off on a subroutine. There the monster's old position is turned back to open path, and a move is made at random (as long as there is path in the direction he is supposed to go).

```
5000  P$(MR,MC) ="1"
5010  X = RND
5020  IF X>.75 THEN 5110
5030  IF X>.5 THEN 5090
5040  IF X>.25 THEN 5070
5050  MC =MC -1 - (P$(MR,MC-1)="")
5060  GOTO 5120
5070  MC = MC+1+(P$(MR,MC+1)="")
5080  GOTO 5120
5090  MR=MR-1-(P$(MR-1,MC)="")
5100  GOTO 5120
5100  MR=MR+1+(P$(MR+1,MC)="")
5120  P$(MR,MC)="6"
5130  RETURN
```

Notice how the lines that make the moves also check that the move is possible, and cancel any attempts to walk through walls. In practice, this routine will quite often leave the monster in the same position.

CALL HCHAR lines can be worked into the subroutine so that the monster is displayed on the screen. When he moves, the path behind him can be left clear, or blacked out again as you wish.

Variations

1 Ghosts. As everybody knows, ghosts can walk through walls. This particular talent is very useful to the games programmer, as it means that the parts of the lines that check the path ahead can be simply left out. Hurray, an easy variation!

2 Hungry Horrors on the Hunt. You can make your monster more threatening by having him head straight for the hero. This has a useful side effect of producing a simpler routine. The monster's position is compared with the hero's, and then adjusted to bring it closer. The routine would look something like this:

```
5000  P$(MR,MC) ="1"
5010  R1 = MR -(MR<R)+(MC>C)       (R,C the
5020  C1 = MC -(MC<C)+(MC>C)       hero's
5030  IF P$(R1,C1)="" THEN 5060    co-ordinates)
5040  MR = R1
5050  MC = C1
5060  P$(MR,MC) ="6"
5070  RETURN
```

Here's what this routine does in two typical situations.

Line	Case 1 Monster (5,5) Hero (7,2)	Case 2 Monster (10,3) Hero (8,8)
5010	R1 = MR+1 = 6	R1 = MR−1 = 9
5020	C1 = MC−1 = 4	C1 = MC+1 = 4
5030	P$(6,4) = "1" (path)	P$(9,4) = "" (wall)
5040	MR =6	} these lines jumped
5050	MC = 4	
5060	P$(6,4) ="6"	P$(10,3) ="6"
Result	Gets closer	No move

Introducing those two temporary stores, R1 and C1, means that the original monster co-ordinates are left alone, and only changed if a move is possible. You don't have to do it this way, but the alternative is rather complicated 'value of truth' lines.

Because this routine does not let the monsters walk through walls, your hero has some chance of escape. If your monsters are ghosts, then he could find life very dangerous. You had better equip him with some means of defending himself!

If the effect is still too terrifying for your players, then introduce a random factor. Instead of a simple command to make the monster move:

```
IF M=1 THEN...        (off to move routine)
```

use a line like this:

```
IF (M=1)*(RND>.5) THEN...
```

Now the monster will stay where he is half the time.

Special note for cheats

Those limits that you use in random lines do not have to be fixed. That last line could read:

```
IF (M=1)*(RND>RL) THEN...
```

RL, the Random Limit is given a value early on in the program:

```
RL = .5     (or whatever limit you want)
```

You then write in a routine to ask 'WHO'S THERE?' and include after it this type of routine:

```
IF N$<>"HONEST SID"THEN...     (jump the next
RL = .8                         line)
```

This resets the Random Limit only for 'Honest Sid', and only you know the password. If you give yourself too much of an edge people might start to wonder why you keep winning, and they might decide to examine your program.

You are far too honest for that sort of thing, aren't you, so let's get back to our monsters, but first . . . 'Compute a Grimble'.

You can now adapt your Grimble program so that the 99 moves the Grimble. Give the Grimble a target – his home, and have his movements directed towards it. Make sure that it checks the path ahead for Grimble cages. If one is there, the Grimble should head off in another (random) direction.

Multiple monsters

These can be managed in exactly the same way as single

monsters, except that now you use arrays rather than simple variables, and each of the monster routines must be enclosed in a loop.

Bring them all to life at the beginning:

```
FOR N=1 TO 4
M(N)=1
NEXT N
```

Give them all a position:

```
FOR N =1 TO 4
MR(N) = INT(RND*18)+3
MC(N) = INT(RND*18)+3
IF P$(MR(N),MC(N)) ="" THEN ...     (back and
P$(MR(N),MC(N)) ="6"                 try again)
NEXT N
```

And so on for the other routines. Simply add (N) after each of the monster variables. Here we are assuming that 4 monsters are enough for any hero, but you can have as many as you like. You just change the numbers at the start of the loop. The more you use, the slower the program will run, but speed is not usually important in this sort of game.

11

Colour changing

One of the 99's useful features is the way that it lets you change the colour of characters that are already on the screen. We can develop a number of games out of this facility.

Have you ever come across those timed light switches? You sometimes find them in the stairwells of blocks of flats. You press the switch and the light stays on for a couple of minutes. It then turns itself off automatically. We could fit a 'light switch' into a program like 'minefield' (see the chapter on Obstacles). Each time you bump into one of the scattered blocks, the screen will light up and show you where the blocks are. You will have time to get a quick look at the field before it all disappears again. The object of the game now is to see how few times you bump into things on your way across the screen. Here is the basis of this type of game:

```
 10 RANDOMIZE
 20 SC =0        (score)
 30 CALL CHAR(128,“FFFFFFFFFFFFFFFF”)
                                         The obstacle block)
 40 CALL COLOR(13,1,1)     (made transparent)
 50 CALL CLEAR
 60 FOR N=1 TO 50
 70 R= INT(RND*24)+1
 80 C = INT(RND*32)+1      (scatters 50 blocks)
 90 CALL HCHAR(R,C,128)
100 NEXT N
110 R=1
120 C=1          (player's start point)
130 CALL HCHAR(R,C,42)
140 CALL SOUND(250,330,1)
150 CALL KEY(3,K,S)
160 IF S=0 THEN 150
170 R=R-(K=88)+(K=69)
180 R=R-(R=0)+(R=25)          (check line)
190 C=C-(K=68)+(K=83)
200 C=C-(C=0)+(C=33)
210 CALL GCHAR(R,C,Z)
220 IF Z<>128 THEN 290          (jump if free space
                                 ahead)
230 CALL COLOR(13,2,1)         (blocks coloured black)
240 CALL SOUND(1000,440,1)       (this gives you 2
250 CALL SOUND(1000,880,1)       seconds to look)
260 CALL SOUND(1,-1,1)
270 CALL COLOR(13,1,1)         (blocks invisible again)
280 SC=SC +1
290 IF (R=24) * (C=32) THEN 310       (the end
300 GOTO 130                           at last?)
310 PRINT “SCORE =”;SC
320 INPUT “AGAIN ? ”:A$
330 IF A$ =“Y” THEN 10
340 END
```

Type this in and try it. A score of less than 4 is pretty good. You can adjust the difficulty of the game by changing the numbers of blocks that are printed by the loop starting at line 60, and also by reducing the sound times in lines 240 and 250.

Variations

1 Have two types of obstacles. One type will be 'light switches', the other type will be mines. Define the characters differently, so that when the light goes on you can spot the mines, and just hope that a light is the first thing you bump into!
2 Have several types of obstacles – each with a different point value. Again, it should be clear when the lights go on just how much each is worth.
3 Back to the start. When the player bumps into a block and the lights are turned on, reset his position and send him

back to the start. Leave the obstacles alone though, so that the player can gradually learn his way through. This game could get quite frustrating, especially when chance has thrown a lot of blocks in the bottom right hand corner.

4 More and more. Start with fewer blocks on the screen – 20 should be about right, and then add another set each time the player bumps into a lightswitch. Now each collision makes the game more difficult. Combine this with a Back to the start game if you want to make life really hard.

12
Time and place

1 Timed inputs

There will be times when you will want to allow your players only a limited time in which to respond to a question, or problem. The standard INPUT line will wait forever, so that is no use. You can, however, build a timer into a CALL KEY routine. If you write this in as a subroutine, it can be used whenever you want it in your main program. This is the basic form it will take:

```
1000 C=0        (Count)
1010 CALL KEY(3,K,S)
1020 C=C+1
1030 IF C>20 THEN 1070      (timed input loop)
1040 IF S = 0 THEN 1010
1050 PRINT K
1060 RETURN
1070 PRINT "TOO SLOW"
1080 RETURN
```

This particular routine can be worked up into a game to test reaction times. Instead of writing a fixed limit in the Count check line, you make it variable. Each time the player reacts quickly enough, his limit is reduced. A 'Too slow' response leads to an increased time limit. The object of the game is to get the lowest possible time limit. In the program outlined below the problem is to press a letter chosen at random by the 99. The game could be expanded into a two-player version, in which case the input loop would need to be enclosed in a further loop, and two Count stores used.

```
FOR P=1 TO 2
CALL KEY(P,K,S)
C(P)=C(P)=1
...
```

Here's the flowchart. There is a check program at the end of the chapter.

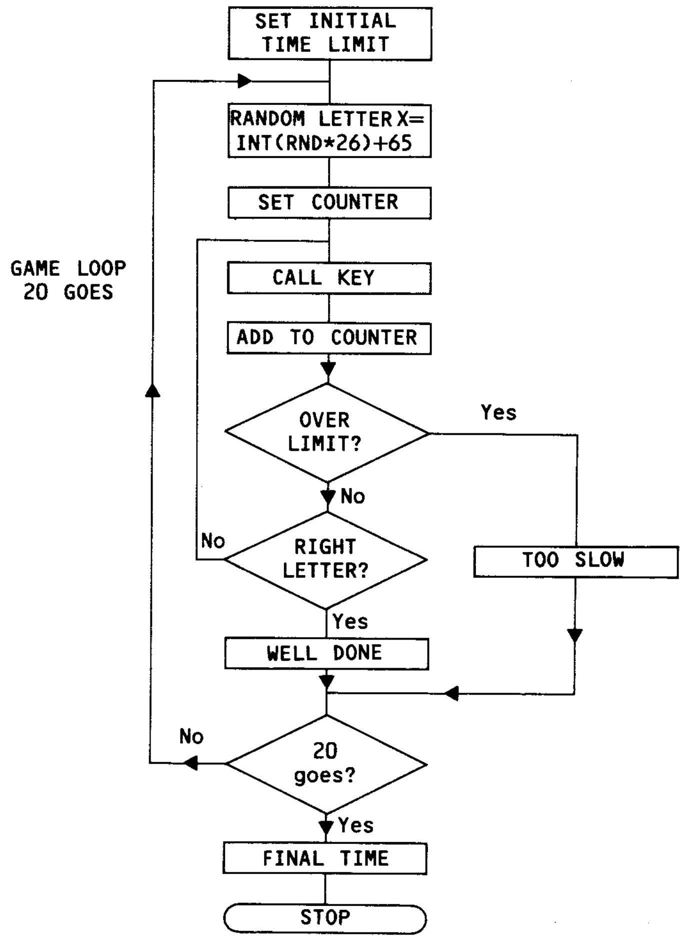


Figure 42

2 Input anywhere

You clearly cannot use a normal INPUT line in games where it is important that the screen is not disturbed. However, a CALL KEY line will only take in one keystroke, and will not print the character. If your player must enter a word or a number of more than one digit, then you need a special routine. The example below shows how you can do this:

```
 10 T$="TEST"
 20 A$=""
 30 CALL CLEAR
 40 C=5
 50 CALL HCHAR(10,C,63)      (prints a question
 60 CALL KEY(3,K,S)           mark at 10,5)
 70 IF S=0 THEN 60
 80 IF K=13 THEN 130       (13 is ENTER)
 90 CALL HCHAR(10,C,K)       (prints the letter)
100 A$=A$&CHR$(K)
110 C=C+1
120 GOTO 50
130 IF A$=T$ THEN 160
140 PRINT "WRONG"
150 STOP
160 PRINT "RIGHT"
170 STOP
```

The player's answer is printed across the screen, starting from 10,5. That question mark in line 50 is so that he can see where he is. The inclusion of a CALL SOUND line would help to catch the player's attention. Notice how the letters are gathered into the A$ store by line 100. Without this you would not be able to check the total answer.

This could be made into a subroutine, with ENTER as the signal to return to the main program, where A$ would be checked against the required answer.

3 Yes or no?

Where you want your users to give a yes/no reply, or select an option, then make sure that all unacceptable replies are ignored:

```
INPUT "AGAIN ?(Y/N)":A$
IF A$ ="Y" THEN...
IF A$ ="N" THEN ...
GOTO
```

This would also ignore any replies written in small type. You may remember from Starter Pack 1, that a CALL KEY(3. . . line resets the keyboard so that the 99 sees all letters in large capitals.

The check lines also ignore 'YES' and 'NO' replies. A slight alteration will cover this:

```
IF SEG$(A$,1,1) ="Y" THEN...
```

Now it checks only the first letter of the A$ input. Used with a CALL KEY line, this routine now accepts "Y", "Y","YES" and "YES". The extra effort on your part will make life easier for your users.

4 Numbers only

The normal INPUTs have built in checks to prevent people typing letters into number stores. Your Input Anywhere routine does not, yet. If you use it to collect a number reply, and try and evaluate the number using VAL(A$) the program will crash if your user has typed in a letter by mistake. The following routine checks through the string, character by character, and warns the user if any non-number character is used.


```
1000 INPUT A$
1010 FOR V=1 TO LEN(A$)
1020 IF SEG$(A$,V,1)>"9" THEN 1060
1030 NEXT V
1040 PRINT VAL(A$)
1050 GOTO 1000
1060 PRINT "INVALID ANSWER"
1070 GOTO 1000
```

Type it in and see. The routine can be adapted into a subroutine for regular use.

5 Print anywhere

You will have come across this if you have read Starter Pack 2. It is included here for the benefit of those of you who have not.

This routine will print anything anywhere you like on the screen. You will find it, in several slightly different forms, in many of the programs on the tape, normally at 6000. The main program specifies the string to be printed (W$), and the Row and Column start points (R1,C1), before it jumps to the subroutine.

```
6000 FOR Q = 1 TO LEN(W$)
6010 CALL HCHAR(R1,C1+Q,ASC(SEG$(W$,Q,1)))
6020 NEXT Q
6030 RETURN
```

A CALL SOUND line can be included in the routine to give a 'teletype' effect.

Speed game check program

```
10  T= 25
20  FOR N = 1 TO 20
30  X = INT(RND*26)+65
40  PRINT CHR$(X)
50  C = 0
60  CALL KEY(3,K,S)
70  C = C+1
80  IF C>T THEN 130
90  IF K<>X THEN 60
100 PRINT "WELL DONE"
110 T = T-1
120 GOTO 150
130 PRINT "TOO SLOW"
140 T = T+1
150 NEXT N
160 PRINT "FINAL TIME ";T
170 STOP
```


Appendices A Program LISTS

You may find it useful to compare the lists with the programs while they are running, as this can help to make some techniques clearer. For a more detailed look at any particular program, set BREAKPOINTS before you run. The use of TRACE commands is not recommended here, as the constant stream of line numbers will almost certainly destroy the screen layout, and make it even more difficult to follow the program.

TARGET

```
10 REM  TARGET
20 REM  MACBRIDE 1983
30 CALL SCREEN(8)
40 CALL CLEAR
50 PRINT TAB(10);"TARGET"::
60 PRINT "  THIS SHOWS HOW A BUL
LET "::
70 PRINT "  FIRING ROUTINE WORKS
."::
80 PRINT "  MOVE THE ""GUN"" USI
NG "::
90 PRINT "  >>S<< TO GO LEFT"::
100 PRINT "  >>D<< TO GO RIGHT."
::
110 PRINT "  PRESS >>F<< TO FIRE
.":::
120 PRINT "  THERE IS A PROGRAM
INDEX "::
130 PRINT "  AT THE END.":::
140 PRINT "  PRESS ANY KEY TO BE
GIN "
150 CALL SOUND(500,250,1)
160 CALL KEY(3,K,S)
170 IF S=0 THEN 160
180 CALL CLEAR
190 REM  graphics
200 CALL CHAR(128,"00003098FEFF1
830")
210 REM  128 = flying plane
220 CALL CHAR(129,"1038383838002
844")
230 REM  129 = bullet
240 CALL CHAR(130,"20646C38F80C0
600")
250 REM  130 = falling plane
260 CALL CHAR(131,"10101010387C7
C7C")
270 REM  131 = gun
280 GC=15
290 REM  Gun Column at start
300 F=0
310 REM  gun not firing (f=0)

320 BR=20
330 REM  Bullet Row at start
340 PRINT "    PRESS >>Q<< TO QU
IT"
350 FOR TC=1 TO 32
360 CALL HCHAR(5,TC,128)
370 IF F=1 THEN 540
380 REM  key check

390 CALL KEY(3,K,S)
400 REM   quit ? asc(q)=81

410 IF K=81 THEN 800
420 REM   move gun ?
  asc(d)=68  asc(s)+83
430 IF (K=68)+(K=83)=0 THEN 480
440 CALL HCHAR(20,GC,32)
450 REM    "value of truth"
    for gun movement

460 GC=GC-(K=68)+(K=83)
470 REM   fire button pressed
  asc(f)=70
480 IF K<>70 THEN 650
490 REM    fire !!

500 F=1
510 CALL SOUND(50,200,1,-8,1)
520 BC=GC
530 REM   check for hit

540 CALL GCHAR(BR,BC,Z)
550 REM   print bullet

560 CALL HCHAR(BR,BC,129)
570 CALL HCHAR(BR,BC,32)
580 IF Z=128 THEN 710
590 REM   change Bullet Row

600 BR=BR-3
610 IF BR>3 THEN 650
620 REM   reset bullet after
 miss
630 BR=20
640 F=0
650 CALL HCHAR(5,TC,32)
660 CALL HCHAR(20,GC,131)
670 NEXT TC
680 GOTO 350
690 REM   crash routine

700 CALL HCHAR(5,TC,32)
710 FOR N=5 TO 20
720 X=(N-4)/2+(X>30)
730 CALL HCHAR(N,TC+X,130)
740 CALL SOUND(50,-6,1)
750 CALL HCHAR(N,TC+X,32)
760 NEXT N
770 INPUT " ANOTHER GO CRACKSHOT
 ?(Y/N) ":A$
780 IF A$="Y" THEN 140
790 IF A$<>"N" THEN 770
800 CALL SCREEN(16)
810 CALL CLEAR
820 PRINT "       PROGRAM INDEX"
::
830 PRINT "  INSTRUCTIONS.......
....40"::
840 PRINT "  GRAPHICS DEFINITION
...190"::
850 PRINT "  VARIABLES SET .....
...280"::
860 PRINT "  MAIN LOOP.........3
50-680"::
870 PRINT "  KEY CHECKS.........
...380"::
880 PRINT "  GUN MOVEMENT.......
...440"::
890 PRINT "  BULLET ROUTINES....
...490"::
900 PRINT "  CRASH!!............
...690":::
```

RACETRACK

```
10 REM  RACETRACK
15 REM  MACBRIDE 1983
20 REM  Joysticks or keys ?
25 GOSUB 455
30 CALL CHAR(128,"FFFFFFFFFFFFFF
FF")
35 CALL SCREEN(11)
40 CALL CLEAR
45 REM       the track
50 CALL HCHAR(1,3,128,28)
55 CALL HCHAR(20,3,128,28)
60 CALL VCHAR(2,3,128,19)
65 CALL VCHAR(2,30,128,19)
70 CALL HCHAR(5,7,128,20)
75 CALL HCHAR(16,7,128,20)
80 CALL VCHAR(6,7,128,10)
85 CALL VCHAR(6,26,128,10)
90 REM     print anywhere
      routine used
95 W$="GRAND PRIX"
100 R=10
105 C=11
110 GOSUB 555
115 REM     car graphics

120 CALL CHAR(130,"EE44FEFFFE44E
E00")
125 CALL CHAR(133,"10BAFEBA38BAF
EBA")
130 CALL CHAR(132,"0077227FFF7F2
277")
135 CALL CHAR(131,"5D7F5D1C5D7F5
D08")
140 REM  car start points
145 CR=18
150 CC=15
155 REM    sound duration      con
trols speed of movement
160 M=400
165 REM  initial direction

170 D=1
175 CALL HCHAR(CR,CC,129+D)
180 CALL SOUND(M,-3,1)
185 REM  Joysticks ?

190 IF J=1 THEN 235
195 REM   key controls

200 CALL KEY(3,K,S)
205 REM    speed change

210 M=M-(10*(K=88))+(10*(K=69))
215 REM    direction change

220 D=D+(K=83)-(K=68)
225 GOTO 255
230 REM  Joystick controls

235 CALL JOYST(1,X,Y)
240 M=M-(2.5*Y)
245 D=D+(X/4)
250 REM    M must not be 0

255 M=M-(10*(M=0))
260 REM    keep D in range
265 D=D-(4*(D=0))+(4*(D=5))
270 REM      rub out car

275 CALL HCHAR(CR,CC,32)
280 REM    change position

285 CC=CC+(D=3)-(D=1)
290 CR=CR+(D=4)-(D=2)
295 REM  check for crash
300 CALL GCHAR(CR,CC,Z)
305 IF Z=128 THEN 320
310 GOTO 175
315 REM  print "CRASH" in
      right place
320 W$="CRASH"
325 R=CR
330 C=CC+(5*(CC>25))
335 GOSUB 555
340 REM  random noises

345 FOR N=1 TO 10
350 P=RND*50+200
355 CALL SOUND(100,P,1)
360 NEXT N
365 INPUT "ANOTHER GO ?(Y/N)":A$
370 IF A$="Y" THEN 30
375 IF A$<>"N" THEN 365
380 CALL CLEAR
385 PRINT "       PROGRAM INDEX":
:
390 PRINT "  TRACK..............
...45"::
395 PRINT "  GRAPHICS DEFINITION
..115"::
400 PRINT "  VARIABLES SET......
..140"::
405 PRINT "  CONTROLS - KEYS....
..195":
410 PRINT "            - JOYSTICK
S.235"::
415 PRINT "  CAR MOVEMENT.......
..270"::
420 PRINT "  CRASH!!............
..315"::
425 PRINT "  JOYSTICK OR KEYS?..
..450"::
430 PRINT "  PRINT ANYWHERE....
..550"::
435 PRINT " SEE ""CHANGING DIREC
TIONS""."
440 STOP
445 REM       sub-routines
450 REM    Joysticks or keys?
455 INPUT "DO YOU WANT TO USE JO
YSTICKS (Y/N) ?":A$
460 PRINT ::::
465 J=0
470 IF (A$="Y")+(A$="y")THEN 485
475 IF (A$="N")+(A$="n")THEN 505
480 GOTO 455
485 J=1
490 PRINT "PLEASE CHECK THAT JOY
STICKS ARE PLUGGED IN."::
495 PRINT "ALPHA LOCK MUST BE OF
F (UP).":: 
500 GOTO 530
505 PRINT "  YOUR CONTROLS .....
"::
510 PRINT "  STEER LEFT......S":
:
515 PRINT "  STEER RIGHT.....D":
:
520 PRINT "  ACCELERATOR.....E":
:
525 PRINT "  BRAKE..........X":
:
530 PRINT "   PRESS ANY KEY TO B
EGIN "
535 CALL KEY(3,K,S)
540 IF S=0 THEN 535
545 RETURN
550 REM  print anywhere

555 FOR Q=1 TO LEN(W$)
560 CALL HCHAR(R,C+Q,ASC(SEG$(W$
,Q,1)))
565 NEXT Q
570 RETURN
```

MAZE

```
10 REM  MAZE
20 REM  MACBRIDE 1983
30 CALL CLEAR
40 PRINT TAB(13);"MAZE"::::
50 PRINT "  THIS PROGRAM SHOWS H
OW A"::
60 PRINT " PATH-MAKING ROUTINE W
ORKS"::
70 PRINT " AND HOW AN ARRAY CAN
BE "::
80 PRINT " USED TO MAP A MAZE.":
::
90 PRINT " PRESS ANY KEY TO BEGI
N":::
100 CALL SOUND(500,250,1)
110 CALL KEY(3,K,S)
120 IF S=0 THEN 110
130 X1=.35
140 X2=.85
150 REM  x1,x2 set limits for ra
ndom moves.
160 SR=1
170 REM  SR Start Row
180 SC=1
190 REM  Start Column
200 FR=10
210 REM   Finish Row
220 FC=10
230 REM  Finish Column
240 CALL CHAR(128,"FFFFFFFFFFFF
FFF")
250 CALL SCREEN(8)
260 CALL CLEAR
270 FOR R=1 TO 10
280 FOR C=1 TO 10
290 CALL HCHAR(R,C+5,48)
300 CALL HCHAR(R,C+20,128)
310 NEXT C
320 NEXT R
330 W$="   . THE ARRAY       THE
MAZE"
340 L=12
350 GOSUB 6000
355 CALL SOUND(500,250,1)
360 W$="    PRESS ANY KEY TO GO O
N"
370 L=24
380 GOSUB 6000
390 CALL KEY(3,K,S)
400 IF S=0 THEN 390
410 R=SR
420 C=SC
430 W$="  Variables. Row=  : Col
umn= "
435 L=16
440 GOSUB 6000
445 W$="              X(RND)="
450 L=14
455 GOSUB 6000
460 W$="     RND CHECK      LINE I
N USE "
465 L=18
470 GOSUB 6000
475 W$="    PRESS ANY KEY TO SEE
MOVE"
480 L=24
485 GOSUB 6000
490 CALL SOUND(500,250,1)
495 CALL SOUND(100,250,1)
500 CALL KEY(3,K,S)
510 IF S=0 THEN 500
520 REM  new move
530 FOR N=1 TO 3
540 CALL HCHAR(R,C+5,32)
550 CALL HCHAR(R,C+20,32)
560 CALL SOUND(10,200,1)
570 CALL HCHAR(R,C+5,49)
580 CALL HCHAR(R,C+20,42)
590 CALL SOUND(10,300,1)
600 NEXT N
610 REM  path-making routine   I
gnore the GOSUBS - they    produ
ce the comments.
620 X=RND
630 IF X>X2 THEN 820
640 IF X>.5 THEN 770
650 IF X>X1 THEN 720
660 REM  down a row
670 R=R+1+(R=10)
680 W$="    X<"&STR$(X1)&"
 670 R=R+1+(R=10)"
690 GOSUB 5800
700 GOTO 870
710 REM
720 REM  up a row
730 R=R-1-(R=1)
740 W$="  X>"&STR$(X1)&"& X<.5
  730 R=R-1-(R=1)"
750 GOSUB 5800
760 GOTO 870
770 REM  right
780 C=C+1+(C=10)
790 W$="  X>.5 & X<"&STR$(X2)&"
 780 C=C+1+(C=10)"
800 GOSUB 5800
810 GOTO 870
820 REM  left
830 C=C-1-(C=1)
840 W$="    X>"&STR$(X2)&"
 830 C=C-1-(C=1)  "
850 GOSUB 5800
860 REM
870 REM  check for finish
880 IF (C=FC)*(R=FR)THEN 900
890 GOTO 495
900 W$="        !! OUT AT LAST !!
  "
905 L=22
910 GOSUB 6000
915 CALL HCHAR(R,C+5,49)
920 CALL HCHAR(R,C+20,42)
925 CALL SOUND(1000,220,1,277,1,
392,1)
930 W$="    PRESS ANY KEY TO GO
ON  "
940 L=24
950 GOSUB 6000
960 CALL SOUND(1000,294,1,370,1,
440,1)
970 CALL KEY(3,K,S)
980 IF S=0 THEN 970
990 REM
1000 CALL CLEAR
1010 PRINT " YOU CAN FIX THE STA
RT AND"::
1020 PRINT "END POINTS YOURSELF
IF YOU"::
1030 PRINT "WOULD LIKE TO.":::
1040 INPUT "LIKE TO RUN IT AGAIN
 ?(Y/N)":A$
1050 IF A$="Y" THEN 1100
1060 IF A$<>"N" THEN 1040
1070 CALL CLEAR
1080 GOTO 1300
1090 REM   user's input
1100 INPUT "FIX YOUR OWN ENDS?(Y
/N)":A$
1110 IF A$="Y" THEN 1140
1120 IF A$="N" THEN 150
1130 GOTO 1100
1140 INPUT "Start Row ?(1 TO 10)
":SR
1150 INPUT "Start Column ?(1 TO
10)":SC
1160 INPUT "Finish Row ?(1 TO 10
)":FR
1170 INPUT "Finish Column ?(1 TO
 10)":FC
1180 REM  adjusts limits for rnd
 check lines
1190 X1=.25-(.1*(FR>SR))+(.1*(FR
<SR))
1200 X2=.75-(.1*(FC>SC))+(.1*(FC
<SC))
1210 GOTO 240
1300 PRINT TAB(8);"PROGRAM INDEX
"::
1310 PRINT "  INTRODUCTION......
....40"::
1320 PRINT "  VARIABLES SET.....
...120"::
1330 PRINT "  PRINT SCREEN......
...250"::
1340 PRINT "  FLASHING ""0"" &""
*""....520"::
1350 PRINT "  PATH-MAKER........
...610"::
1360 PRINT "  CHECK FOR END.....
...870"::
1370 PRINT "  RE-RUN ?.........
..1000"::
1380 PRINT "  PRINT SUB-ROUTINES
..5800":::
1390 STOP
5800 L=20
5810 GOSUB 6000
5820 R$=STR$(R)&" "
5830 FOR N=1 TO LEN(R$)
5840 CALL HCHAR(16,17+N,ASC(SEG$
(R$,N,1)))
5850 NEXT N
5860 C$=STR$(C)&" "
5870 FOR N=1 TO LEN(C$)
5880 CALL HCHAR(16,28+N,ASC(SEG$
(C$,N,1)))
5890 NEXT N
5900 X$="0"&STR$(X)
5910 FOR N=1 TO 4
5920 CALL HCHAR(14,17+N,ASC(SEG$
(X$,N,1)))
5930 NEXT N
5940 RETURN
6000 FOR Q=1 TO LEN(W$)
6010 CALL HCHAR(L,Q,ASC(SEG$(W$,
Q,1)))
6020 NEXT Q
6030 RETURN
```

DUEL

```
10 REM  DUEL
20 REM  MACBRIDE 1983
30 CALL SCREEN(8)
40 CALL CLEAR
50 PRINT TAB(13);"DUEL":::
60 PRINT " THIS IS GIVEN AS AN E
XAMPLE"::
70 PRINT " OF A TWO-PLAYER ACTIO
N GAME"::
80 PRINT " WRITTEN IN TI BASIC."
:::
90 PRINT " IT HAS ROUTINES FOR B
OTH "::
100 PRINT " JOYSTICK AND KEY CON
TROLS."::
110 PRINT " LIST THE GAME AFTER
YOU "::
120 PRINT " HAVE FINISHED PLAYIN
G AND"::
130 PRINT " SEE HOW IT WORKS."::
:
140 CALL KEY(3,K,S)
150 INPUT " ARE YOU USING JOYSTI
CKS ?   (Y/N) ":A$
160 J=0
170 REM  Joystick indicator
180 IF A$="Y" THEN 280
190 IF A$<>"N" THEN 150
200 PRINT ::
210 PRINT "            CONTROLS":
220 PRINT "  LEFT              R
IGHT"::
230 PRINT "  >E< TO MOVE  TANK
>I<"::
240 PRINT "  >S< TO STEER LEFT
>J<"::
250 PRINT "  >D< TO STEER RIGHT
>K<"::
260 PRINT "  >F< TO FIRE BULLET
>L<":::
270 GOTO 330
280 J=1
290 PRINT " THE ALPHA LOCK MUST
BE OFF"::
300 PRINT " PUSH FORWARD TO GO."
::
310 PRINT " STEER LEFT OR RIGHT"
::
320 PRINT " PRESS ORANGE BAR TO
FIRE"::
330 REM   graphics
       left tank
340 CALL CHAR(128,"001F187EFF7E3
C")
350 CALL CHAR(129,"3838383838101
010")
360 CALL CHAR(130,"00F8187EFF7E3
C")
370 CALL CHAR(131,"1010103838383
838")
380 REM  bullet
390 CALL CHAR(132,"0000081C08000
000")
400 REM  right tank
410 CALL CHAR(136,"001F187EFF7E3
C")
420 CALL CHAR(137,"3838383838101
010")
430 CALL CHAR(138,"00F8187EFF7E3
C")
440 CALL CHAR(139,"1010103838383
838")
450 REM  bullet
460 CALL CHAR(140,"0000081C08000
000")
470 REM  edge
480 CALL CHAR(144,"FFFFFFFFFFFF
FFF")
490 REM  wall block
500 CALL CHAR(145,"FFC3B59999B5C
3FF")
510 PRINT "   PRESS ANY KEY TO B
EGIN":::
520 CALL KEY(3,K,S)
530 IF S=0 THEN 520
540 REM   screen edges
550 CALL COLOR(15,5,9)
560 CALL COLOR(13,16,1)
570 CALL SCREEN(3)
580 CALL CLEAR
590 CALL HCHAR(1,3,144,29)
600 CALL HCHAR(20,3,144,29)
610 CALL VCHAR(2,3,144,18)
620 CALL VCHAR(2,31,144,18)
630 REM   "walls"
640 RANDOMIZE
650 FOR N=1 TO 25
660 BR=INT(RND*16)+3
670 BC=INT(RND*23)+7
680 W=INT(RND*6)+2
690 IF RND>.5 THEN 730
700 IF BC+W>28 THEN 660
710 CALL HCHAR(BR,BC,145,W)
720 GOTO 750
730 IF BR+W>18 THEN 660
740 CALL VCHAR(BR,BC,145,W)
750 NEXT N
760 REM  set tank positions    a
nd directions
770 R(1)=19
```

```
780 C(1)=4
790 D(1)=1
800 R(2)=2
810 C(2)=30
820 D(2)=3
830 REM   direction (D)
840 REM   1=right,2=down
850 REM   3=left,4=up
860 REM  GAME STARTS HERE
870 FOR P=1 TO 2
880 CALL HCHAR(R(P),C(P),119+8*P
+D(P))
890 NEXT P
900 IF J=0 THEN 1110
910 REM  joystick control

920 FOR P=1 TO 2
930 CALL JOYST(P,X,Y)
940 IF (X=0)*(Y=0)THEN 1030
950 CALL HCHAR(R(P),C(P),32)
960 REM  change direction

970 D(P)=D(P)+X/4
980 D(P)=D(P)-(4*(D(P)=0))+(4*(D
(P)=5))
990 IF Y<4 THEN 1020
1000 REM  move tank

1010 GOSUB 1420
1020 CALL HCHAR(R(P),C(P),119+8*
P+D(P))
1030 IF F(P)THEN 1080
1040 REM  fire?

1050 CALL KEY(P,K,S)
1060 IF S=0 THEN 1080
1070 GOSUB 1370
1080 NEXT P
1090 GOTO 1280
1100 REM  key controls

1110 FOR P=1 TO 2
1120 CALL KEY(P,K,S)
1130 IF S=0 THEN 1260
1140 REM  fire?

1150 IF (K=12)*(F(P)=0)THEN 1250
1160 CALL HCHAR(R(P),C(P),32)
1170 REM  change direction

1180 D(P)=D(P)-(K=3)+(K=2)
1190 D(P)=D(P)-(4*(D(P)=0))+(4*(
D(P)=5))
1200 IF K<>5 THEN 1230
1210 REM  move tank
1220 GOSUB 1420
1230 CALL HCHAR(R(P),C(P),119+8*
P+D(P))
1240 GOTO 1260
1250 GOSUB 1370
1260 NEXT P
1270 REM  tank firing ?

1280 FOR P=1 TO 2
1290 IF F(P)=0 THEN 1310
1300 GOSUB 1520
1310 NEXT P
1320 GOTO 860
1330 REM  end of main loop

1340 REM  sub-routines from
     here down
1350 REM
1360 REM    shell start point
     and direction
1370 F(P)=D(P)
1380 SR(P)=R(P)
1390 SC(P)=C(P)
1400 RETURN
1410 REM  tank mover

1420 R1=R(P)
1430 C1=C(P)
1440 R1=R1-(D(P)=2)+(D(P)=4)
1450 C1=C1-(D(P)=1)+(D(P)=3)
1460 CALL GCHAR(R1,C1,Z)
1470 IF Z>143 THEN 1500
1480 R(P)=R1
1490 C(P)=C1
1500 RETURN
1510 REM   shell in flight

1520 FOR N=1 TO 6
1530 SR(P)=SR(P)-(F(P)=2)+(F(P)=
4)
1540 SC(P)=SC(P)-(F(P)=1)+(F(P)=
3)
1550 REM  check ahead

1560 CALL GCHAR(SR(P),SC(P),Z)
1570 IF (Z)143-8*P)*(Z<148-8*P)T
HEN 1690
1580 IF Z>143 THEN 1640
1590 CALL HCHAR(SR(P),SC(P),124+
8*P)
1600 CALL SOUND(10,-5,1)
1610 CALL SOUND(1,-1,1)
1620 CALL HCHAR(SR(P),SC(P),32)
1630 GOTO 1650
1640 N=6
1650 NEXT N
1660 F(P)=0
1670 RETURN
1680 REM  hit.H=player who
          has been Hit
1690 H=2+(P=2)
1700 F(P)=0
1710 FOR N=1 TO 5
1720 FOR T=1 TO 4
1730 CALL HCHAR(R(H),C(H),119+8*
H+T)
1740 CALL SOUND(50,-T,1)
1750 NEXT T
1760 NEXT N
1770 REM   end or carry on?

1780 RESTORE 1690
1790 FOR R2=21 TO 23
1800 READ W$
1810 FOR Q=1 TO LEN(W$)
1820 CALL HCHAR(R2,Q+3,ASC(SEG$(
W$,Q,1)))
1830 NEXT Q
1840 NEXT R2
1850 DATA "PRESS >Q< TO QUIT","
 >S< TO START AGAIN","  >C< TO C
ARRY ON"
1860 CALL KEY(3,K,S)
1870 IF S=0 THEN 1860
1880 CALL HCHAR(21,1,32,96)
1890 IF K=81 THEN 1940
1900 IF K=83 THEN 550
1910 IF K=67 THEN 900
1920 GOTO 1780
1930 REM   end of game
1940 CALL CLEAR
1950 PRINT TAB(8);"PROGRAM INDEX
"::
1960 PRINT "   INTRODUCTION......
....40"::
1970 PRINT "   GRAPHICS..........
...330"::
1980 PRINT "   SCREEN LAY-OUT....
...540"::
1990 PRINT "   VARIABLES SET.....
...760"::
2000 PRINT "   START OF GAME LOOP
...860"::
2010 PRINT "   CONTROLS -JOYSTICK
...910"
2020 PRINT "             -KEYS....
..1100"
2030 PRINT "   SUB-ROUTINES"
2040 PRINT "     START SHELL.....
..1360"
2050 PRINT "     MOVE TANK.......
..1410"
2060 PRINT "     FIRE AND HIT?...
..1510"
2070 PRINT "   SPINNING TANK.....
..1680"
2080 PRINT "   END?..............
..1770"
2090 STOP
```

BAT

```
10 REM  BAT
20 REM  MACBRIDE 1983
30 REM  graphics
40 FOR N=1 TO 10
50 READ G$
60 CALL CHAR(127+N,G$)
70 NEXT N
80 REM  bats
90 DATA 0F2F7F3EFEFEFCE0
100 DATA F0F4FE7C7F7F3F07
110 DATA 073F7F7F7CFEF4F0
120 DATA E0FCFEFE3E7F2F0F
130 REM  cave mouth
140 DATA 3C7EFFFFFFFF7E3C
150 REM  edges
160 DATA FFFFFFFFFFFFFFFF
170 DATA FFFFFFFFFFFFFFFF
180 DATA FFFFFFFFFFFFFFFF
190 DATA FFFFFFFFFFFFFFFF
195 DATA FF818181818181FF
200 CALL SCREEN(6)
205 CALL KEY(3,K,S)
210 CALL CLEAR
220 PRINT TAB(11);"BAT"
230 PRINT "  TRY TO KNOCK THE BA
T ";CHR$(129)::
240 PRINT "  INTO THE CAVE ";CHR
$(132)::
250 PRINT "  USING THE SPECIALLY
 "::
260 PRINT "  DESIGNED BAT-KNOCKE
R ";CHR$(137)
265 PRINT TAB(24);CHR$(137):::
270 INPUT "  ARE YOU USING JOYST
ICKS ? (Y/N)":A$
280 J=0
290 CALL KEY(3,K,S)
300 IF A$="Y" THEN 330
310 IF A$="N" THEN 360
320 GOTO 270
330 J=1
340 PRINT "  PLEASE CHECK THAT A
LPHA"::
350 PRINT "  LOCK IS OFF (UP)"::
:
360 PRINT "  PRESS ANY KEY TO BE
GIN"::
370 CALL KEY(3,K,S)
380 IF S=0 THEN 370
390 CALL CLEAR
400 REM   screen layout
410 KR=13
420 KC=5
430 REM  Bat Knocker start
  points
440 CALL HCHAR(1,3,133,20)
450 CALL HCHAR(20,3,135,20)
460 CALL VCHAR(2,3,136,18)
470 CALL VCHAR(2,22,134,18)
480 CALL HCHAR(10,12,132)
485 CALL VCHAR(KR,KC,138,2)
490 GOSUB 1000
500 REM  1000 -  prints and mov
es bat knocker
510 W$="TRY MOVING THE BAT-KNOCK
ER"
520 C=3
530 R=22
540 GOSUB 6000
550 W$=" PRESS >>G<< TO START GA
ME"
560 R=24
570 GOSUB 6000
580 IF J=1 THEN 700
585 RESTORE 600
590 FOR N=1 TO 5
600 READ W$,R
610 C=22
620 GOSUB 6000
625 NEXT N
630 DATA "  STEERING",2
640 DATA ">S< LEFT",6
650 DATA ">D< RIGHT",8
660 DATA ">E< UP",10
670 DATA ">X< DOWN",12
700 CALL KEY(3,K,S)
710 IF K=71 THEN 750
720 GOSUB 1000
730 GOTO 700
750 REM  game starts here
760 D=INT(RND*4)+1
770 REM  direction
780 BR=INT(RND*8)+2
790 BC=INT(RND*15)+5
800 REM  Bat start point
810 CALL HCHAR(BR,BC,127+D)
815 REM  knocker move?

820 CALL KEY(3,K,S)
830 GOSUB 1000
835 REM  rub-out bat

840 CALL HCHAR(BR,BC,32)
845 REM  move bat

850 BR=BR+(D<3)-(D>2)
860 BC=BC+(D=1)+(D=4)-(D=2)-(D=3
)
865 REM  what's ahead ?

870 CALL GCHAR(BR,BC,Z)
872 REM  space - fly on

875 IF Z=32 THEN 800
878 REM  cave-mouth end

880 IF Z=132 THEN 1250
885 REM  bat-knocker?

890 IF Z<>137 THEN 930
900 D=D-(D=1)-(D=3)+(D=2)+(D=4)
910 D=D+(4*(D=5))-(4*(D=0))
915 BC=BC+(D=1)+(D=4)-(D=3)-(D=2
)
920 GOTO 810
930 REM  edge routine
940 E=Z-132
950 D=D+1-(2*(D=E))
960 D=D+(4*(D<4))
970 BR=BR-(BR=1)+(BR=20)
980 BC=BC-(BC=3)+(BC=22)
990 GOTO 810
1000 REM  knocker print/move
1005 IF (J=0)*(S=0)THEN 1140
1010 IF J=1 THEN 1060
1020 CALL VCHAR(KR,KC,32,2)
1030 REM  key controlled
1040 KR=KR-(K=88)+(K=69)
1050 KC=KC-(K=68)+(K=83)
1055 GOTO 1100
1060 REM  joystick
1070 CALL JOYST(1,X,Y)
1075 IF (X=0)*(Y=0)THEN 1140
1080 CALL VCHAR(KR,KC,32,2)
1085 KR=KR-Y/4
1090 KC=KC+X/4
1095 REM  check for edge

1100 KR=KR-(KR(2)+(KR)18)
1110 KC=KC-(KC(4)+(KC)21)
1120 CALL VCHAR(KR,KC,137,2)
1130 CALL HCHAR(10,12,132)
1140 RETURN
1250 CALL SOUND(1000,500,1)
1260 CALL SOUND(1000,750,1)
1270 INPUT "ANOTHER GAME ?(Y/N)"
:A$
1280 IF A$="Y" THEN 360
1290 IF A$<>"N" THEN 1270
1300 CALL SCREEN(8)
1310 CALL CLEAR
1320 PRINT TAB(8);"PROGRAM INDEX
"::
1330 PRINT "  GRAPHICS..........
....30"::
1340 PRINT "  INTRODUCTION......
...210"::
1350 PRINT "  SCREEN LAYOUT.....
...400"::
1360 PRINT "  GAME STARTS HERE..
...750"::
1370 PRINT "  BAT MOVEMENT......
...800"::
1380 PRINT "  EDGE ROUTINE......
...930"::
1390 PRINT "  KNOCKER MOVEMENT..
..1000"::
1400 PRINT "  PRINT ANYWHERE....
..6000"::
1410 STOP
6000 FOR Q=1 TO LEN(W$)
6010 CALL HCHAR(R,C+Q,ASC(SEG$(W
$,Q,1)))
6020 NEXT Q
6030 RETURN
```

DRAGON

```
10 REM  DRAGON
20 REM  MACBRIDE 1983
25 CALL SCREEN(8)
30 CALL CLEAR
40 PRINT TAB(12);"DRAGON":::
50 PRINT "  THERE'S GOLD TO BE F
OUND"::
60 PRINT " AND DRAGONS AND GOBLI
NS TO"::
70 PRINT "  TACKLE AS YOU WORK YO
UR WAY"::
80 PRINT "  THROUGH THE DRAGON'S
LAIR,":::
90 PRINT "  YOU WON'T KNOW WHERE
 THEY"::
100 PRINT "  ARE UNTIL YOU MEET
THEM.":::
110 PRINT "  THE ARROW KEYS (E,S
,D,X)"::
120 PRINT "  WILL MOVE YOUR MAN.
"::
130 PRINT "  PRESS ANY KEY TO BE
GIN"
140 CALL KEY(3,K,S)
150 IF S=0 THEN 140
160 PRINT :::
165 RANDOMIZE
170 PRINT "  I AM PREPARING A PA
TH FOR"::
180 PRINT "  YOU - IT WON'T TAKE
 LONG."::
185 REM   set up array

190 OPTION BASE 1
200 DIM P$(21,21)
205 REM  start main path

210 R=2
220 C=2
230 GOSUB 4000
235 REM  reached end?

240 IF (R=20)+(C=20)THEN 260
250 GOTO 230
255 REM  four false trails

260 FOR T=1 TO 4
270 R=T*3
280 C=16-R
290 FOR N=1 TO 20
300 GOSUB 4000
310 IF (R=20)+(C=20)THEN 330
320 GOTO 340
330 N=20
340 NEXT N
350 NEXT T
355 REM   Armour Alters    the
odds for dragon-slaying

360 A=1
365 REM   our Money

370 M=INT(RND*10)*100+100
380 PRINT " YOU HAVE";M;"GOLD CO
INS"::
390 PRINT "  A SWORD AND SHIELD W
ILL "::
400 PRINT " HELP IF YOU MEET A D
RAGON."::
410 INPUT "LIKE A SWORD ? ONLY 1
00 GOLD PIECES.(Y/N)":A$
415 PRINT ::
420 IF A$="Y" THEN 440
430 GOTO 460
440 M=M-100
450 A=A*.8
460 IF M<100 THEN 520
470 INPUT "HOW ABOUT A NICE SHIE
LD ?     ONLY 100 COINS.(Y/N)":A
$
475 PRINT ::
480 IF A$="Y" THEN 500
490 GOTO 520
500 M=M-100
510 A=A*.8
520 PRINT "  ONE MOMENT PLEASE"
525 REM  15 incidents

530 FOR T=1 TO 15
540 R=INT(RND*18)+2
550 C=INT(RND*18)+2
560 IF P$(R,C)="" THEN 540
570 X=RND
580 IF X>.7 THEN 640
590 IF X>.4 THEN 620
600 P$(R,C)="2"
610 GOTO 650
620 P$(R,C)="3"
630 GOTO 650
640 P$(R,C)="4"
650 NEXT T
655 REM  graphics
   black block
660 CALL CHAR(128,"FFFFFFFFFFFF
FFF")
665 REM  the hero

670 CALL CHAR(129,"18183030C60705
0D8")
675 REM  straws

680 CALL CHAR(136,"34242C3C34242
C3C")
690 CALL CLEAR
695 CALL COLOR(14,3,8)
700 CALL SCREEN(2)
705 GOSUB 3000
710 P$(2,2)="1"
720 P$(20,20)="5"
725 CALL HCHAR(1,1,128,768)
730 W$="OUT"
740 C1=21
```

```
750 R1=20
760 GOSUB 6000
770 W$="GOLD "
780 C1=23
790 R1=1
795 GOSUB 6000
800 GOSUB 5910
802 W$="DAMAGE"
804 R1=5
806 GOSUB 6000
808 CALL HCHAR(7,24,48)
810 R=2
820 C=2
830 REM  player start point
840 CALL HCHAR(R,C+1,129)
850 CALL SOUND(100,400,1)
860 CALL KEY(3,K,S)
870 IF S=0 THEN 860
880 FOR N=22 TO 32
890 CALL VCHAR(8,N,128,12)
900 NEXT N
910 CALL HCHAR(R,C+1,32)
915 REM  move hero

920 IF K=88 THEN 1070
930 IF K=69 THEN 1030
940 IF K=83 THEN 990
950 C=C+1
960 IF P$(R,C)<>"" THEN 1170
970 C=C-1
980 GOTO 1100
990 C=C-1
1000 IF P$(R,C)<>"" THEN 1170
1010 C=C+1
1020 GOTO 1100
1030 R=R-1
1040 IF P$(R,C)<>"" THEN 1170
1050 R=R+1
1060 GOTO 1100
1070 R=R+1
1080 IF P$(R,C)<>"" THEN 1170
1090 R=R-1
1100 CALL HCHAR(R,C+1,129)
1110 W$="!WALL!"
1120 C1=23
1130 R1=9
1140 CALL SOUND(500,200,1)
1150 GOSUB 6000
1160 GOTO 840
1170 V=VAL(P$(R,C))
1175 REM  what's ahead?
 1=path, 2=gold,3=dragon  ,. 4=g
oblins, 5=out.
1180 ON V GOSUB 1200,1250,1350,2
000,2600
1190 GOTO 840
1200 RETURN
1240 REM  ** gold **

1250 W$="MORE GOLD"
1260 R1=9
1270 CALL SOUND(500,750,1)
1280 GOSUB 6000
1290 G=10+INT(RND*10)*10
1300 W$=STR$(G)&" COINS"
1310 R1=11
1320 GOSUB 6000
1330 GOSUB 5900
1335 P$(R,C)="1"
1340 RETURN
1350 RESTORE 1350
1355 REM  ** dragon! **

1360 X=25+INT(RND*30*A)
1370 G=10+INT(RND*10)*10
1380 C1=22
1390 R1=9
1395 W$="!!DRAGON!!"
1397 CALL SOUND(1000,500,1,-8,1)
1400 GOSUB 5990
1405 W$="  HE HAS  "
1410 GOSUB 5990
1415 W$=STR$(G)&" COINS "
1420 GOSUB 5990
1425 W$="SCORE OVER"
1430 GOSUB 5990
1435 W$=STR$(X)&" TO KILL"
1440 GOSUB 5990
1445 W$=" FIGHT OR "
1450 GOSUB 5990
1455 W$="   RUN?   "
1460 GOSUB 5990
1465 W$=">F< OR >R<"
1470 GOSUB 5990
1475 CALL KEY(3,K,S)
1480 IF K=82 THEN 1500
1485 IF K=70 THEN 1730
1490 GOTO 1475
1500 IF RND>.7 THEN 1600
1510 IF RND>.5 THEN 1550
1520 CALL SOUND(500,523,1)
1530 B$="ESCAPED"
1540 GOTO 1680
1550 CALL SOUND(500,220,1)
1560 A$="ESCAPED"
1570 B$="WOUNDED"
1580 D=D+2
1590 GOTO 1650
1600 CALL SOUND(500,-4,1)
1610 A$="YOU DROPPED"
1620 B$="YOUR GOLD"
1630 M=0
1640 REM  display
1650 W$=A$
1660 R1=18
1665 C1=23
1670 GOSUB 6000
1680 W$=B$
1690 R1=19
1700 GOSUB 6000
1710 GOSUB 5800
1720 GOTO 1900
1730 W$="SCORE"
1740 R1=18
1750 GOSUB 6000
1760 Y=INT(RND*80)
1770 FOR N=1 TO Y
1775 N$=STR$(N)
1780 FOR T=1 TO LEN(N$)
1790 CALL HCHAR(18,27+T,ASC(SEG$
(N$,T,1)))
1795 NEXT T
1800 CALL SOUND(10,100+20*N,1)
1810 NEXT N
1815 CALL SOUND(1000,1000,1)
1820 IF Y<X THEN 1870
1825 FOR R1=17 TO 19
1830 READ W$
1835 GOSUB 6000
1840 NEXT R1
1842 DATA "HE'S DEAD","YOU GET
","HIS GOLD "
1845 CALL SOUND(500,262,1,330,1,
392,1)
1850 CALL SOUND(500,262,1,330,1,
392,1)
1855 GOSUB 5900
1857 P$(R,C)="1"
1860 GOTO 1920
1870 W$="WOUNDED!"
1880 R1=18
1882 D=D+2
1884 GOSUB 5800
1890 GOSUB 6000
1900 CALL SOUND(1000,466,1)
1910 IF D>6 THEN 1930
1920 RETURN
1930 W$=" YOU ARE DEAD.BUT DON'T
 FEEL"
1935 C1=2
1940 R1=22
1950 GOSUB 6000
1960 W$="TOO BURNED UP ABOUT IT.
"
1970 R1=23
1980 GOSUB 6000
1990 GOTO 2650
2000 G=10+INT(RND*10)*10
2001 REM  ** goblins **

2004 CALL CHAR(144,"383A127E7878
286C")
2006 CALL COLOR(15,13,16)
2010 CALL SOUND(500,440,1)
2020 CALL SOUND(500,220,1)
2025 CALL HCHAR(R,C+1,144)
2030 R1=9
2040 W$="!!GOBLIN!!"
2050 C1=22
2060 GOSUB 6000
2070 W$=" PAY "&STR$(G)
2080 GOSUB 5990
2090 W$="OR GAMBLE?"
2100 GOSUB 5990
2110 W$="  PRESS   "
2120 GOSUB 5990
2130 W$=">P< OR >G<"
2140 GOSUB 5990
2150 CALL KEY(3,K,S)
2160 IF K=71 THEN 2220
2170 IF K=80 THEN 2190
2180 GOTO 2150
2190 M=M-G
2200 GOSUB 5910
2210 RETURN
2220 W$=" PICK THE "
2230 GOSUB 5990
2240 W$="LONG STRAW"
2250 GOSUB 5990
2260 W$="  ■    ■  "
2270 GOSUB 5990
2280 GOSUB 5990
2290 W$=">L< OR >R<"
2300 GOSUB 5990
2310 CALL SOUND(500,500,1)
2320 CALL KEY(3,K,S)
2330 IF (K=76)+(K=82)=0 THEN 2
0
2340 IF RND>.5 THEN 2390
2350 W$="  ■       "
2360 GOSUB 6000
2370 IF K=76 THEN 2490
2380 GOTO 2430
2390 W$="       ■  "
2400 GOSUB 6000
2410 IF K=82 THEN 2490
2420 REM  wrong guess
2430 W$=" YOU LOST "
2440 GOSUB 5990
2450 CALL SOUND(1000,110,1)
2460 M=M-G*2
2470 GOSUB 5910
2480 RETURN
2490 W$=" YOU WIN "
2500 GOSUB 5990
2510 CALL SOUND(1000,550,1)
2520 M=M+G*2
2530 GOSUB 5910
2540 RETURN
2600 W$="SUCCESS AT LAST!"
2601 REM  ** out **

2605 CALL HCHAR(R,C+1,129)
2610 C1=5
2620 R1=22
2630 GOSUB 6000
2640 CALL SOUND(1000,262,1,330,1
,392,1)
2650 W$="PRESS ANY KEY TO GO ON"
2660 R1=24
2670 GOSUB 6000
2680 CALL KEY(3,K,S)
2690 IF S=0 THEN 2680
2700 INPUT "LIKE ANOTHER GAME ?(
Y/N) ":A$
2710 IF A$="Y" THEN 2900
2720 IF A$<>"N" THEN 2700
2730 CALL SCREEN(8)
2740 CALL CLEAR
2750 PRINT TAB(8);"PROGRAM INDEX
"::
2760 PRINT "  INTRODUCTION......
....40"
2770 PRINT "  MAZE DRAWER.......
...165"
2780 PRINT "  SWORD AND SHIELD..
...355"
2790 PRINT "  SCATTER INCIDENTS.
...525"
2800 PRINT "  GRAPHICS..........
...655"
2810 PRINT "  START SCREEN......
...690"::
2820 PRINT "  GAME START........
...830"
2830 PRINT "  MOVEMENT .........
...910"
2840 PRINT "  INCIDENTS"
2850 PRINT "         MORE GOLD..
..1250"
2860 PRINT "         DRAGON!!...
..1350"
2870 PRINT "         GOBLINS....
..2000"::
2880 PRINT "  PATH-MAKER........
..4000"
2890 PRINT "  MESSAGE PRINTING..
..5800"
2895 STOP
2900 FOR T=1 TO 21
2910 FOR N=1 TO 21
2920 P$(T,N)=""
2930 NEXT N
2940 NEXT T
2950 GOTO 10
3000 FOR S=1 TO 13
3010 CALL COLOR(S,2,8)
3020 NEXT S
3030 RETURN
4000 X=RND
4010 IF X>.8 THEN 4130
4020 IF X>.5 THEN 4100
4030 IF X>.3 THEN 4070
4040 R=R+2+(2*(R>19))
4050 P$(R-1,C)="1"
4060 GOTO 4150
4070 R=R-2-(2*(R<3))
4080 P$(R+1,C)="1"
4090 GOTO 4150
4100 C=C+2+(2*(C>19))
4110 P$(R,C-1)="1"
4120 GOTO 4150
4130 C=C-2-P-(2*(C<3))
4140 P$(R,C+1)="1"
4150 P$(R,C)="1"
4160 RETURN
5800 W$=STR$(D)
5810 R1=7
5815 C1=23
5820 GOSUB 6000
5830 GOTO 5910
5900 M=M+G
5910 W$=STR$(M)
5920 R1=3
5925 C1=23
5930 CALL SOUND(500,600,1)
5940 GOTO 6000
5990 R1=R1+1
6000 FOR Q=1 TO LEN(W$)
6010 CALL HCHAR(R1,C1+Q,ASC(SEG$
(W$,Q,1)))
6020 NEXT Q
6030 RETURN
```

B
Sprites and TI EXTENDED BASIC

The EXTENDED BASIC module is not particularly cheap, but it does offer a number of very valuable facilities to the games programmer. Of these the most important for action games are those routines which operate SPRITES.

Sprites are characters which can be placed on the screen anywhere, and moved smoothly in any direction. The sprites can change colour, size, shape, speed or position while they are in use. Additional subprograms can be used to check for collisions or to find the locations of sprites, or the distance between two sprites. Sprites can move more than twice as quickly as a character that is running through an HCHAR loop, and they move just as quickly whether they are tiny sprites taking up only one character space, or huge ones that use sixteen spaces. If you have ever tried to move a multi-character graphic across the screen, you will appreciate how valuable this is.

The smoothness of movement of the sprites comes from the use of a high-resolution screen. Instead of their positions being set on a 32 by 24 character space grid, a fine grid 192 dot-rows by 256 dot-columns is used. The sprite is automatically rubbed out as it moves, and its movement is set by giving a row and column velocity. The effect is to allow smooth movement in any direction, forwards, backwards, up, down or at any angle. (figure 43)

This single line is all you need to start a sprite off.

```
CALL SPRITE(#1,96,16,20,20,0,60)
```

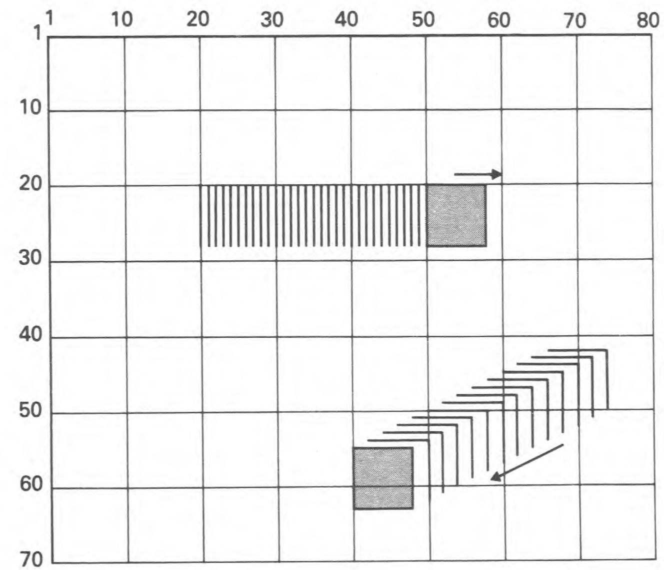



Figure 43

That line makes sprite #1 take on the shape of character 96, colours it white(16), sets it down at 20,20, and gives it a velocity of 0 rows and 60 columns. This means it moves across the screen to the right. When it reaches the edge it is whipped round to the other side automatically. Compare this with the number of lines needed to get the same effect using only TI BASIC.

Sprites can be magnified. A normal (single character) sprite can be blown up so that it occupies 4 spaces. (figure 44)

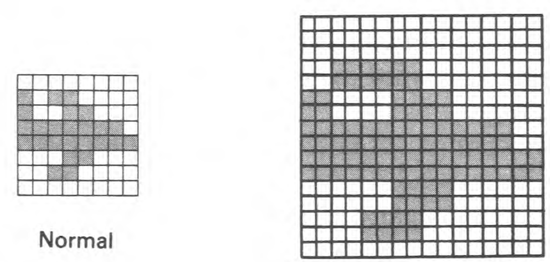
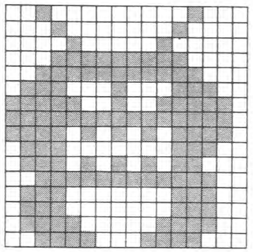


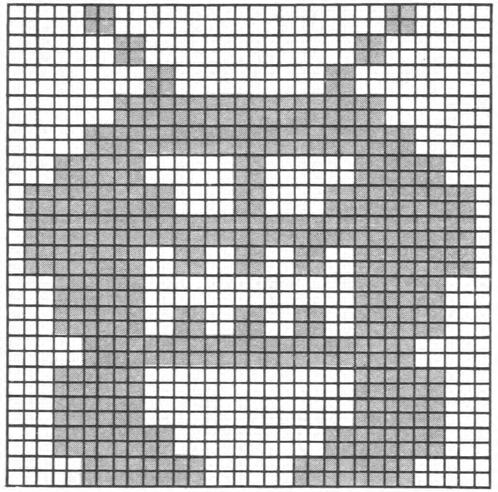
Figure 44

Larger sprites can be created by defining a block of four character squares. These can be further enlarged so that they occupy 16 squares. (Figure 45)

4-character SPRITE (Super-Grimble)



Normal



Magnified

Figure 45

The SPRITE range of subprograms will not take your games up to arcade speeds – only machine code programming can achieve that – but they will allow you fast, smooth action, and make programming easier.

TI EXTENDED BASIC has many other useful features that make for more efficient programming. It is essential if you wish to use the SPEECH SYNTHESISER – which makes the 99 talk! – or if you want to get into Assembly Language programming.

The extra commands and statements of EXTENDED BASIC include ACCEPT AT, which works as an 'Input Anywhere' routine, and DISPLAY AT which allows for printing anywhere. A set of subprograms (ON BREAK, ON WARNING, ON ERROR) cope with these keyboard entries that can cause program crashes in TI BASIC. Finally, EXTENDED BASIC allows the use of multi-statement lines.

```
IF A$= B$ THEN PRINT "WELL DONE":
GOSUB 5000 : GOTO 350 ELSE PRINT"WRONG":
GOSUB 4000: GOTO 370
```

A line like this is possible – not very elegant, but possible. Multi-statement lines can make life much easier than having to jump to separate little routines.