COLLISION COURSE

In the first two instalments of the project we designed routines to set up the scenario for our Minefield game. Now, we look at the control of movement from the keyboard, and inspect the parts of the program that detect collisions between the player characters and the mines.

BBC BASIC has no fewer than four commands that respond to a single keypress. The choice of command will obviously depend on the desired usage. INKEY$ and INKEY are normally used when you want to wait a certain length of time for a possible keypress before carrying on with the rest of the program; GET$ and GET on the other hand will always halt program execution until a key is pressed. These last two commands tend to be used when a response to a question is required, such as 'Another Game Y/N?'. If GET or GET$ is used then the program will wait for an answer. In this case, the only acceptable answers are 'Y' and 'N'. We can use a loop to REPEAT the GET instruction UNTIL the answer is 'Y' or 'N', as follows:

```
1000 PRINT "ANOTHER GAME Y/N?"
1010 REPEAT
1020    A$=GET$
1030 UNTIL A$="Y"OR A$="N"
1040 Etc
```

If GET$ or INKEY$ is used then the key pressed is interpreted as its character string, as in the above example. If we use GET or INKEY then a numeric rather than a string value is returned; this value is the ASCII code of the key pressed. These options allow the programmer to test for keys that do not have a corresponding character, such as the Return or cursor keys. The statement *FX4,1 may be used to make the cursor keys return ASCII codes. If this is done, the keys have the values:

Left cursor	136
Right cursor	137
Down cursor	138
Up cursor	139

Let us say that we wished to accept only left and right cursor inputs to our program. The following program segment uses INKEY to wait for a quarter of a second for an input:

```
1000 *FX4,1:REM TURN ON CURSOR ASCII MODE
1010 REPEAT
1020    A=INKEY(25)
1030 UNTIL A=136 OR A=137
1040 *FX4,0:REM RESTORE CURSOR TO EDIT MODE
```

The parameter 25 in line 1020 tells the computer to wait 25 hundredths of a second before going on with the program.

These statements do not test the keyboard itself but affect an area of memory inside the computer called the keyboard buffer. This is a temporary storage space for characters that are input from the keyboard, and is rather like a cinema queue. New characters typed in tag on to the end of the queue and the processor takes characters from the front of the queue. In this way, if you type in characters faster than the processor can handle them, they are not lost but just wait their turn in the keyboard buffer. As INKEY, GET, INKEY$ and GET$ normally test the front of the keyboard buffer queue you have no way of knowing how long a particular character has been sitting in the buffer waiting to be processed. In games that are controlled from the keyboard this can make for a sluggish response, as the program may be processing earlier key presses whilst the player is making new ones. For example, if you fill the keyboard buffer with cursor-right codes then all of these will need processing before the program can respond to a cursor-left command. This can leave the player frantically pressing the cursor-left button and wondering why the object being controlled is still moving right!

There are two solutions to this problem. The first is always to clear out the keyboard buffer just prior to testing it. This can be done using the statement *FX15,1. Alternatively, we can use a further variation of INKEY. As described above, INKEY( ) waits for a length of time, specified by the number in the bracket, for a key to be pressed before continuing with the program. We can, however, make INKEY test the keyboard instead of the keyboard buffer by specifying a *negative* number in the brackets following the command. Each key has a negative number assigned to it for this purpose, a full list of which is given on page 275 of the User Guide. In our program, we shall use the cursor keys to control movement. The values of the keys to be used with INKEY are:

Cursor-left	-26
Cursor-right	-122
Cursor-down	-42
Cursor-up	-58

The following procedure uses INKEY to test the keyboard directly for each of the four cursor keys in turn. If one of the keys is being pressed then another procedure ('move') is accessed, with two parameters being passed. These parameters hold information about the direction in which the mine