15 A Worked Example of a Video Game

In this chapter we examine the use of both BASIC and Assembler programlning for animated video games. In general games are expensive, and the players interest can be short-lived. If players can achieve a reasonable result themselves then most would prefer to write their own simple video games, and spend their money only on good-quality sophisticated games. Tape cassette 2 contains an extensive example of the sort ofgame that most competent programmers can reasonably expect to write, without going to the extreme of putting the whole game into machine code. The game itself, *RON the ROBOT in the WASTE of TYME*, is a typical 'shoot-em up' game; however the techniques we discuss here could just as easily be applied to 'bat and ball' (such as TENNIS) or 'tactical' games (such as PAC-MAN). For you to make the most of the explanations in this chapter we advise you to get the companion cassette tapes, and LOAD and RUN these program listings.

The most important part of writing any game is the planning. It is helpful to draw a plan of the intended screen on graph-paper: sketch in the proposed size and positions of fixed objects (the background) and the areas to be traversed by moving objects (the foreground). Both background and foreground are usually composed of user-defined characters. This should fix a scale for the objects and will give a general impression of the final game. It is worth spending some time at this stage to ensure that the proposed action can actually fit into the graphics area. Once you have decided on a screen layout you must create the multi-coloured objects for the foreground and the background by using the CHARACTER GENERATOR 2 of chapter 5. Before writing any complex code for moving these characters about you must lirst place them on the screen, both singly aiid in groups, in order to see what impression they make. We saw that the fastest way of putting a large number of characters on the screen is to 'prynt' them (see chapter 5). In our program we shall adjust 'prynt' so that it draws strings of characters down the screen.

Exercise 15.1

Alter the CHARACTER GENERATOR 2 program to allow characters to be defined in neighbouring blocks of two or four, which will allow compound shapes to be constructed with ease. This option will need the INPUT of two or

four ASCII codes to tell the program where to store the multiple character data.

Always conceive and write your game programs as modules. Such subtasks can be programmed separately either in BASIC or directly in assembly code, and tested individually before they are combined into larger modules or the final program. Speed is the essence, so that most of the BASIC procedures should be translated into machine code for the final version of the game. There are, however, some sections of the game that do not need speed, such as any explanatory text before the game starts and some initialisation processes. These can be left in BASIC.

Since we are aiming at speed of execution we must make all routines as explicit as possible. We calculate every possible value at the programming stage or by a previously run initialising program. The barest minimum of computation should be done while the game is being played.

One important technique for minimising the calculation, which we use in the example program, is the cascade or *multiple entry point method*. We write cascade routines which are called by a pointer, each cascade being made up of separate sections. Only one section is entered per execution of the routine. Inside the routine the pointer may be changed so that with the next entry a different section is obeyed. This continues with the pointer usually moving down through the sections of the cascade until it reaches the bottom, where it will usually be reset to the top. Such cascades are normally implemented by GOSUB with variable labels in BASIC, or as indirect jumps or conditional branching statements in assembly language. Normally this technique is used on several cascades which are called from within a loop. This loop describes the major tasks needed in playing the game. It gives us the ability to carry on different processes in an interwoven yet independent manner. Apparent parallelism of this sort is essential for games, where independent events may be following complicated courses. Effectively the program operates on two or more routines simultaneously, with one section only from each cascade being executed each time through the loop.

Consider the following two simple programs run in mode 7, which perform independent functions. Listing 15.1 waits until a key is pressed and then shoots a point across the screen. Listing 15.2 continuously moves a cross up the screen in a zig-zag pattern. Both programs use the fast animation techniques that are found in the game that follows, so it is necessary to slow them down with the *FX,19 command (except on OS 0.1) and REPEAT loops. Excess speed is sometimes a problem in assembler programs, but only when programs are very simple. We combine these two programs into a simple loop of cascades. We set pointers (‘cross’ and ‘point’) to the top of the cascades and control entry to the corresponding cascades by changing their values inside the routines (listing 15.3). In this simple game you type any key and the point moves across the screen: when the point and cross coincide – SPLAT.

Listing 15.1

```
200 REM dot
209 REM fire dot across screen when a is pressed
210 MODE 7 : VDU23,1,0;0;0;0;
220 PRINT TAB(0,11);"."
230 A$=GET$
240 D=0
250 PRINT TAB(D,11);" " : D=D+1
260 IF D=40 THEN 220
270 PRINT TAB(D,11);"."
279 REM wait for start of frame to slow movemen down
280 *FX19
290 GOTO 250
```

Listing 15.2

```
300 REM cross
309 REM continually move cross up screen
310 MODE 7 : VDU 23,1,0;0;0;0;
320 X=8 : Y=23
330 PRINT TAB(X,Y);"+"
340 PRINT TAB(X,Y);" " : Y=Y-1
350 IF Y<0 THEN 320
359 REM make cross move in zig-zag
360 IF Y>12 THEN X=X+1 ELSE X=X-1
370 PRINT TAB(X,Y);"+"
379 REM slow down movement of cross
380 T=TIME : REPEAT UNTIL TIME>T+2
390 GOTO 340
```

Listing 15.3

```
100 REM main loop
110 dot=200 : cross=300 : D=0
120 MODE 7 : VDU23,1,0;0;0;0;
129 REM run modified dot and cross programs at same time.
130 REPEAT
140 GOSUB dot : GOSUB cross
150 UNTIL FALSE

200 REM dot cascade
210 PRINT TAB(0,11);"." : dot=220 : RETURN
219 REM now use INKEY$ to check for keypress so cross can move.
220 IF INKEY$(0)="" THEN RETURN
230 D=0 : dot=240 : RETURN
240 PRINT TAB(D,11);" " : D=D+1
250 IF D=40 THEN dot=210 : RETURN
260 PRINT TAB(D,11);"."
270 RETURN

300 REM cross cascade
310 X=8 : Y=23
320 PRINT TAB(X,Y);"+" : cross=330 : RETURN
330 PRINT TAB(X,Y);" " : Y=Y-1
340 X=X+1 : PRINT TAB(X,Y);"+"
```

```
350 IF Y<=11 THEN cross=380
360 T=TIME : REPEAT UNTIL TIME=T+2
370 RETURN
379 REM as cross changes direction check for dot hitting cross.
380 IF Y=11 AND X=D THEN PRINT TAB(X,Y);"SPLAT!" : END
390 PRINT TAB(X,Y);" " : Y=Y-1
400 IF Y<0 THEN cross=310 : RETURN
410 X=X-1
420 PRINT TAB(X,Y);"+"
430 T=TIME : REPEAT UNTIL TIME=T+2
440 RETURN
```

Exercise 15.2

Add a line to the calling loop to reset the pointers to the top of the cascades and continue the game after a SPLAT, perhaps printing the score (number of hits). Write a 'duck-shoot' game with a hunter who moves left and right under keyboard control while shooting with a shotgun at ducks that are flying across the screen.

How to Write a Game

For a simple game as outlined in the above example (and the worm game of chapter 1) the saving in time and programming effort from using the cascade technique is significant. We shall now describe a good approach for writing a non-trivial game by using the aforementioned ideas, with RON as an illustration. First define and plan your game carefully.

The definition

In our game you are RON, the last robot remaining after a nuclear disaster and you are trying to rescue the few surviving microcomputers, the BEEBS. While doing this you must avoid the Mutant Typists (MUTTS) who are trying to stop you. RON is equipped with a ray gun which he can fire in any of four directions (up, down, left, right), although this will not save him if the MUTTS get their MITTS on him. RON is currently rescuing BEEBS in the WASTE of TYME where there are many pools of radioactive materials that are fatal to MUTTS and RON alike, although RON can vapourise these with his ray gun.

The gun has four firing directions that are specified by pressing 's' for left, 'd' for down, 'f' for right and 'e' for up. If a blast from the gun hits either the MUTTS or the pools they will disappear; however if the MUTTS touch RON he dies. The movement of RON is similarly controlled by 'j', 'k', 'l' and 'i'. A great deal of time at the planning stage of a game should be given to making the controls usable, and the keys you chose must be easy to reach. (If you have joysticks then things are much simpler.)

Just to make things tricky there are occasional appearances of antinuclear demonstrators (DEMONs) wearing 'I told you so' tee-shirts. They are intent on shooting RON, since it was wry probably all his fault anyway!

The characters
You need to create single or multiple multi-colour characters for each participant in a game. This game will be run in mode 2 so we use CHARACTER GENERATOR 2 to draw the pools, RON, the MUTTs, the BEEBs and the DEMONs which will all be two characters high. Because we do not want to be limited to mode 2 character blocks, each object will be stored as the equivalent of 5 characters. (All the characters are stored between ASCII codes 65 and 89.) The screen is considered as a coordinate system with *x*-values from 0 to 79 and *y*-values from 0 to 63 (not 0 to 31). We move in half-characters up the screen, hence the need for 5 characters per object: two are used to draw the object in normal character block position, but three are needed if it moves up half a block since it will range over three blocks. The 'prynt' routine has to be changed to move them about in this fashion. The characters will be stored as file RONCHAR.

The initialisation
The characters can go anywhere on the screen, but to start with we would like them to be fairly randomly spread, but without overlapping. So we start with a BASIC program (stored in file XYDATAP) that finds suitable positions fur 128 objects: 32 BEEBs, 32 pools and 64 MUTTs. It draws coloured squares on the screen to show you where they go, yellow for BEEBs, green for pools and magenta for MUTTs. If you do not like these positions run the program again. These are the *x*/*y* coordinates of the objects and are placed in store between locations &1100 and &1400. RON has 16 waves of MUTTs to contend with, each wave containing more than the one before. The positions within a wave are a subset of the total calculated above. The table of data that specifies the number of occurrences of each object in a round is also stored by this program in the same area of store. Finally the program calculates a table of the addresses of the beginning of each screen line and stores this as well. Once all the data have been prepared it is saved as the file XYDATA.

The logo
This section (file BJPRSNT) would probably be written last of all and simply draws a logo for the game on the screen and follows it with a brief description of the rules of the game. There is no need to use machine code, BASIC is more than adequate for this section. When interrupted by pressing 'S' it will load the main game program (a previously constructed machine-code program) and execute it.

The main game program
We finally come to the main game program (file GAME) which consists of a loop of calls to cascade routines:

MBEEB moves one BEEB each time through the loop.
CHKEYS checks the keyboard to find any change in direction of RON's movement or when the ray gun has been fired.

FIRE moves the photon blasts from the gun around the screen.
MMUT moves some of the mutants each time through the loop. Some of them will change direction to chase RON.
Then the flag KILLED checks whether RON has been killed by a MUTT.
MRON moves RON.
Then the flag DEAD checks to see if RON has hit a pool; if so a new RON appears or the game is over. (You have three RONs to start with.)
DEMON tells if a demonstrator is on the screen, in which case we enter the cascade for its movement. It will use the same FIRE routine as RON.
We then move back to the top of the loop and repeat. Naturally there are a few other checks, such as to see if RON has blasted all the MUTTs or if your score is high enough to gain another RON.

In the game we have tried to show an elementary modular way of approaching the programming of video games in a mixture of BASIC and machine code, where large amounts of data are stored in a variety of tables. The positions for everything on the screen are stored in large tables of *x*-coordinates and *y*-coordinates and collisions are detected simply by assuming that the background is all black (zero). Objects are moved by printing them in their new positions and then by obliterating any leftover parts from their previous positions with blanks. The missiles are drawn by calculating the appropriate byte(s), and then by overprinting the data into the byte and out ofthat byte after the shot has passed. Reprinting often takes less time when removal of the old position can be combined with the printing at the new (see the sections where RON is moving vertically). Remember that discretion is the better part of valour and it is far better to check for problems and to cover them up, rather than to rewrite your program and find that another fault has appeared. When trying to remove faults also remember that a brute-force cover-up will probably be quicker than a fancy fault-avoidance routine.

There is no background in this game for the simple reason that it would necessitate the removal of data from the screen prior to printing. This would be essential in order to allow the background to be restored after the object in the foreground had moved on.

By combining these techniques for moving objects and allowing objects to pass each other, displays of very high quality can be made. Of course for the really fast and complex gaines efficient machine-code routines with even greater numbers of look-up tables must be used. However many of the initialisation and instruction routines can still be in BASIC so do not bother trying to produce completely machine-code programs unless you wish to sell your games.

Finally, as your games' programs become more and more interwoven and cross-connected you must try to keep a simple overview of the game. Use sensible variable names, put in plenty of comments during development, and above all, Don't Panic!

Complete Programs

I Listing 15.1. Type any key.
II Listing 15.2. No data required.
III Listing 15.3. Press any key.
IV See description of tape 2 in the appendix for the video game RON.