Pan

PERSONAL COMPUTER

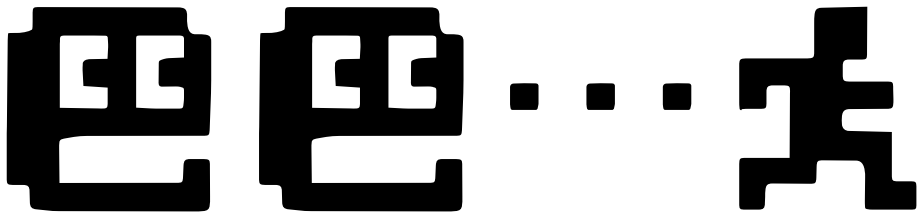
COMPUTER NEWS LIBRARY

JEAN FROST

Instant ARCADE GAMES for the DRAGON 32

Instant Arcade Games
for the
Dragon

Jean Frost

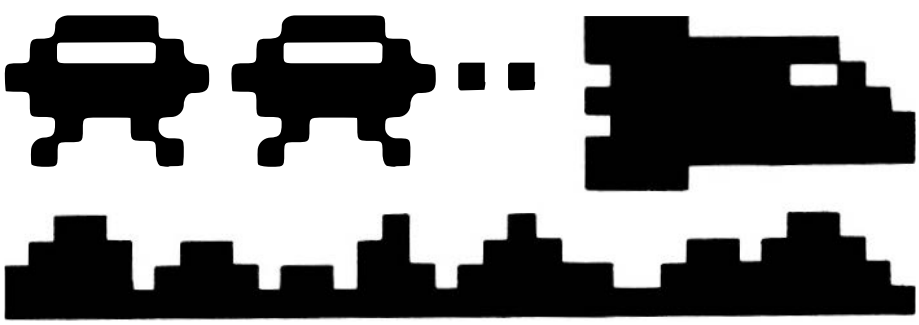


Pan Books London and Sydney

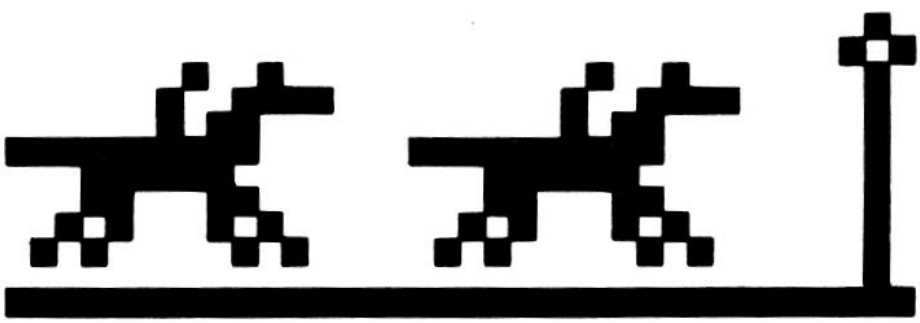
First published 1983 by Pan Books Ltd,
Cavaye Place, London SW10 9PG
in association with Personal Computer News
9 8 7 6 5 4 3 2 1

ISBN 0 330 28271 9

Printed and bound in Great Britain by
Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk



CONTENTS



PREFACE

This book has been designed to help people with little or no programming knowledge construct their own arcade and adventure games. In its simplest form this book is like a Lego-set with ready made pieces that can be put together, even by the complete novice, to form a game. For those who wish to expand their understanding of the BASIC programming language each of the building blocks or routines is explained in straight-forward terms. Further sections of the book explain the routines in greater detail. More experienced programmers may also use this book as a source of ready written and documented routines for inclusion, perhaps with alterations, in their own programs.

Many people are left bewildered when they try to make the transition from buying software to writing their own. Books on BASIC are full of obscure jargon and tend to treat a simple, highly practical subject as though it were an arcane, technical concept. The authors here have attempted to redress the balance by presenting a simple introduction to how a program is put together. A worked example of how to construct a program is given and this can be used as a step by step recipe by the readers for creating their own unique programs. This piece by piece, easy to understand approach allows the reader to become familiar with BASIC programs by 'handling' the routines and using them. In this way each small brick in the structure of programming becomes an understandable everyday object. With continued use these 'objects' will become less mysterious and will gradually become as easy to use as building bricks. In much the same way that children learn to speak English by playing with words, and even alphabet blocks, the reader will assimilate and learn the computing terminology and the BASIC language. This approach allows the learning process to be carried out at your own pace and, hopefully, through the games you construct, at your leisure.

Obviously this approach also requires that this book be used in the same manner as a manual for mending cars or a recipe book: Keep it beside you as you work and refer to it for instructions at the appropriate points. Do not attempt to read straight through the book from cover to cover without keying in and trying many of the routines listed. By the time you have looked at and used most of the example programs you should find your confidence in practical BASIC allows you to write your own programs. We hope you enjoy this book and are rewarded for your studies by a growing understanding of BASIC and many hours of fun.

Jean Frost and Frederick Siviter

CHAPTER 1

BASIC and Games, Computers and Cheesecake

Computer games are becoming a very popular form of entertainment. They are also quite expensive. Many computer owners would like to cut down on this expense by writing their own games and this book is here to help you do just that. How do you write a computer game, then? Well, I think we'd better start by taking a general look at computers and the problems of converting our ideas for games into computer programs.

Computers are really fast operators, and you may think an accountant is pretty nippy on his calculator keyboard, manipulating numbers, but that's just peanuts compared to the speed at which computers work. Unfortunately computers are also dumb. They can't do anything until you tell them what you want doing. So how do we tell a computer to do something? Well, if we really wanted to talk to a computer we would have to speak entirely in numbers, because that's all they really understand! Fortunately most computers are supplied with a built-in interpreter. It's just like the Prime Minister using an interpreter to tell the Russians exactly where they can put their missiles. The BASIC interpreter can translate our words into the number which the computer understands.

Great, so we can just stroll up to the computer, pat it on the keyboard, and say "How much profit did I make selling cheesecake today ?". . . Well it's still not quite that easy, we have to phrase our question very simply and explain everything to the computer in simple and exact terms. The BASIC interpreter can only translate a limited number of words and it assumes that any word it doesn't know is the name of somebody or something. Right, first we need to tell the computer the facts:

The wholesale cost of a cheesecake is 30p each.
The retail price of a cheesecake is 50p each.

We also need to tell the computer things relating to these facts:

Profit is calculated as retail price minus wholesale price.
Total profit is profit per cake multiplied by the number of cheesecakes sold.

Hopefully we have now given the computer all the information a child would need to work out the answer. Of course if we forgot to tell it something the small child might say "you forgot to tell me how many you sold", and many computers would say something dumb like "VARIABLE NOT FOUND" which actually means the same thing. Your DRAGON is much too shy though, and instead of telling you it would assume you hadn't sold any, which is even worse, so we must make sure we tell it everything it needs to know. To tell the computer anything we must express it in BASIC terms. BASIC is like a restricted form of the English language so in BASIC the above statements would be:

```
LET WHOLESALE = 30
LET RETAIL = 50
LET SOLD = 40
```

(That's what we forgot to tell it before.)

```
LET PROFIT = RETAIL - WHOLESALE
LET TOTAL = PROFIT * SOLD
```

(On computers the asterisk '*' is used instead of an x for multiply because we might get the x mixed-up in the middle of some letters.)

The DRAGON, unlike some computers, allows us to miss out the word LET in statements like these, so to save you a lot of extra typing we won't bother with LET from here on. So we can just say:

```
WHOLESALE = 30
```

If you have typed in these lines in exactly as above, pressing ENTER at the end of each line, the computer will have replied 'OK' to each statement you made. The computer now knows what the answer is but, because we didn't ask it to tell us, it just sits there being stupid. To get it to give you the answer tell it to:

```
PRINT TOTAL
```

When you press ENTER the answer 800 will appear.

Let's re-examine how we got the computer to do something. We started with a problem then split the problem up into parts: We gave the computer the facts, we told it what steps it would need to go through to work out the answer, then we told it to tell us the answer. This sequence of instructions, like a recipe, is called an *algorithm* and all programs are just such algorithms (methods) written out in BASIC.

If we wish to use the recipe again we would have to type it in again exactly as before. To save ourselves time we can store the instructions as a program on the computer by numbering each line or statement as we type it in.

```
1 WHOLESALE = 30
2 RETAIL = 50
3 SOLD = 40
4 PROFIT = RETAIL - WHOLESALE
5 TOTAL = PROFIT * SOLD
6 PRINT TOTAL
```

This time as we type in each line the screen will show a LIST of all the instructions we have given it. This is a computer program (program comes from the word programme and, like the ones you get at the theatre, it shows the order in which things are going to be done).

When we want the computer to show us what instructions it has stored we use the BASIC command LIST, followed by ENTER, and the lines of the program will be displayed on the screen once again.

When we want the computer to RUN through the LIST of instructions we simply type the command RUN, then ENTER (which activates the command).

The computer will then do all the instructions in the program and almost instantaneously PRINT the answer on the screen. Well, that was quicker than working it out in your head, but what a time it took getting the question into a state where the computer could answer it. Type in, exactly as shown:

```
3 INPUT "HOW MANY DID YOU SELL";SOLD
```

This will replace the old instruction number 3 and tells the computer to ask you how many cheesecakes you sold each time you RUN the program. If you type anything other than 40 then the computer will give you a different answer. It will always be the right answer and it will be worked out far faster than you could have done it. To save the bother of typing RUN each time we could add an extra instruction to our program:

```
7 GOTO 3
```

This simply tells the computer to GO TO instruction 3 and continue from there down the lines of the program in order. Now we have a program that will keep asking you how many you sold and telling you what profits you would make. You will have to press BREAK to stop the program.

Our program is numbered in consecutive numbers starting at 1. However this is not usually the case with computer programs. Suppose we really had forgotten the line about the number of cheesecakes. We would have typed in a program that looked like this:

```
1 WHOLESALE = 30
2 RETAIL = 50
3 PROFIT = RETAIL - WHOLESALE
4 TOTAL = PROFIT * SOLD
5 PRINT TOTAL
```

If we typed RUN for this set of instructions the computer would assume that SOLD was zero, because no one has told it otherwise. It would therefore multiply PROFIT by zero in line 4 and make TOTAL equal to the result, zero! PRINT TOTAL would therefore print zero. Eventually we would realise what we had forgotten and would want to insert a line to tell it how many cheesecakes it had sold:

```
3 SOLD = 40
```

We would now have to retype all the lines except the first two to create a gap for the new line. Although the computer goes down

the LIST of instructions in order it doesn't care if there are numbers that aren't used. So, we can type our program with line numbers that go up in tens (or any other steps). Now any other lines we've missed out can be given a number between the numbers of the lines already there.

```
10 WHOLESALE = 30
20 RETAIL = 50
30 PROFIT = RETAIL - WHOLESALE
40 TOTAL = PROFIT * SOLD
50 PRINT TOTAL
```

Oops! We forgot that line again, but this time we can simply type

```
25 SOLD = 40
```

and the computer will place it in the LIST of instructions at the appropriate position.

OK, (to quote the computer) we now know roughly what a BASIC program is, so how do we write one that plays a game? We've got to think of the right LIST of instructions. That sounds like a big job to tackle all at once, so we'd better split it up into smaller sub-tasks. Let's start with an overview of the sub-tasks that have to be performed in a typical Arcade-type game. These are:

1. Initialisation. We must first set up the computer so that it is in the correct state to play the game.
2. Instructions. We need to tell the player how to control the movement and firing, etc.
3. Alien graphics. Decide what the opposition looks like.
4. Player graphics. Decide what the good guys look like.
5. Background. Draw some sort of scene against which the action takes place.
6. Set-up routines. Telling the computer the facts.
7. Movement and firing. Move the player and see if a shot is to be fired.
8. Collision detection. See if an alien has been shot or the player has crashed.

9. Explosions. Blow up the poor unsuspecting alien who is very probably a victim of circumstances beyond his (or its) control.

10. Scoring. Award the player some points for his gleeful destructive ability.

11. Fuel and ammunition. Perhaps we might like to affect the quantities of these available to the player.

12. Status display. Show the player how he is getting on.

13. Check for end of game. See if something fatal has happened to the player or if the player has won. We will need to GOTO step 7 if the game isn't over.

14. End of game display. Say goodbye to the player.

Well, that's a lot of things to do. In fact some of them look as if they need large programs just to do one sub-task. BASIC has a pair of commands which allow us to treat small programs as though they are sub-tasks or *subroutines* of a larger program. These commands are GOSUB and RETURN. The first of these is a lot like the GOTO command and makes the computer jump to the line number given, however the GOSUB command also tells the computer to remember where it came from. Once the computer has obeyed the GOSUB command it carries on down the list of instructions in the subroutine until it encounters the RETURN command. At this point it goes back whence it came having performed the sub-task.

Using these commands we can write a control program which simply GOSUBs to the appropriate sub-tasks in the right order. All we will need to do then is define the sub-tasks for it to perform and it will achieve our total task of playing a game. The following listing shows just such a control program constructed on the basis of the need to perform the subroutines as laid out above. This listing and all the other listings in this book have been produced directly from the DRAGON computer. Using the LLIST command, which you need to be Welsh (or slightly BRAHMS & . . .) to pronounce, you can tell the computer to LIST the program on to a printer. This ensures that all the programs in this book are correct, so if you have any trouble check that your typing corresponds exactly with the listings given.

Control Program Listing

```
10 REM ****************
12 REM *INITIALISATION*
14 REM ****************
20 PCLEAR6:PMODE4,3:PCLS
25 NV=0
30 N=43:PG=3:GOSUB 9000
40 DIMFB(1):DIMN(1):DIMAB(1)
50 GM=0:REM *NO. OF GAMES PLAYED*
100 REM **CONTROL PROGRAM**
102
104 REM ***************
106 REM *INSTRUCTIONS*
110 GOSUB 1000
112 REM ******************
119 REM *GRAPHICS(ALIEN)*
120 GOSUB 1100
122 REM ******************
129 REM *GRAPHICS(SELF)*
130 GOSUB 1200
132 REM *************
139 REM *BACKGROUND*
140 GOSUB 1300:GM=GM+1
142 REM ******************
149 REM *START-UP/RESET*
150 GOSUB 1400
152 REM ************
159 REM *MOVE/FIRE*
160 GOSUB 1500
162 REM ****************
169 REM *CHECK FOR HIT*
170 GOSUB 1700
172 REM ************
179 REM *EXPLOSION*
180 IF HIT=1 THEN GOSUB 1800
182 REM *********
```

```
189 REM *SCORING*
190 GOSUB 1900
192 REM ****************
199 REM *FUEL,LASERS ETC*
200 GOSUB 2000
202 REM ****************
209 REM *STATUS DISPLAY*
210 GOSUB 2100
212 REM **************
219 REM *END OF GAME?*
220 GOSUB 2200
222 REM **************
229 REM *ROUND AGAIN*
230 IF FIN=0 THEN 160
232 REM ************
239 REM *GAME OVER*
240 GOSUB 2300
242 REM **************
249 REM *START AGAIN*
250 GOTO 140
252 REM **************
```

As you can see there are a lot of lines in the above program that start with the word REM. This means that anything following the REM is just a REMark or REMinder as to what is going on. The computer ignores these lines completely but stores them in the LISTing so that we can understand what's going on.

In the next chapter there is an example of how to fill in the subroutines for this control program and in the following chapter there is a selection of routines that can be used to make up different games.

However, before going on to that you should carefully type in the two routines and the list of DATA statements given below in listings 1.1, 1.2 and 1.3. These are special routines that will be explained in Chapter 6, but they are important to the working of the program, so they must be entered before going on. This consitutes our first category of initialising the computer to get it ready to play the game.

Listing 1.1

```
9000 PMODE4,PG:SCREEN1,1:REMthis
 is just so you can see it happe
ning
9010 ST=7680+1536*(PG-2)
9020 FOR CH=0 TO N-1:RN=INT(CH/3
2)
9030 FOR Y=0 TO 7:READ CD:IF CD=
999 THEN Y=7:GOTO 9050
9035 IF NV=1 THEN CD=255-CD
9040 POKE ST+224*RN+CH+32*Y,CD
9050 NEXTY,CH
9055 RETURN
```

Listing 1.2

```
9899 REM *PRINT STRING*
9900 IF P$="" THEN RETURN
9910 A$=LEFT$(P$,1):P$=RIGHT$(P$
,LEN(P$)-1)
9920 IF A$=" " THEN YG=144:XG=20
8:GOTO 9950
9930 YG=144:AS=ASC(A$)-65:IF A$<
"A" THEN YG=152:AS=ASC(A$)-48
9940 XG=8*AS
9950 GOSUB 9960:XS=XS+8:GOTO 990
0
9960 PMODE4,3:GET(XG,YG)-(XG+7,Y
G+7),N,G
9970 PMODE4,1:PUT(XS,YS)-(XS+7,Y
S+7),N,PSET:RETURN
```

Listing 1.3

```
500 DATA 0,60,66,66,126,66,66,0
502 DATA 0,124,66,124,66,66,124,
0
504 DATA 0,60,66,64,64,66,60,0
506 DATA 0,120,68,66,66,68,120,0
508 DATA 0,126,64,124,64,64,126,
0
510 DATA 0,126,64,124,64,64,64,0
512 DATA 0,60,66,64,78,66,60,0
514 DATA 0,66,66,126,66,66,66,0
516 DATA 0,62,8,8,8,8,62,0
518 DATA 0,2,2,2,66,66,60,0
520 DATA 0,68,72,112,72,68,66,0
522 DATA 0,64,64,64,64,64,126,0
524 DATA 0,66,102,90,66,66,66,0
526 DATA 0,66,98,82,74,70,66,0
528 DATA 0,60,66,66,66,66,60,0
530 DATA 0,124,66,66,124,64,64,0
532 DATA 0,60,66,66,114,74,60,0
534 DATA 0,124,66,66,124,68,66,0
536 DATA 0,60,64,60,2,66,60,0
538 DATA 0,254,16,16,16,16,16,0
540 DATA 0,66,66,66,66,66,60,0
542 DATA 0,66,66,66,66,36,24,0
544 DATA 0,66,66,66,90,102,66,0
546 DATA 0,66,36,24,24,36,66,0
548 DATA 0,130,68,40,16,16,16,0
550 DATA 0,126,4,8,16,32,126,0
552 DATA 0,0,0,0,0,0,0,0
562 DATA 999,999,999,999,999
564 DATA 0,24,36,44,52,36,24,0
566 DATA 0,8,24,8,8,8,28,0
568 DATA 0,24,36,8,16,32,60,0
570 DATA 0,24,36,24,4,36,24,0
572 DATA 0,8,24,40,72,124,8,0
```

```
574 DATA Ø,6Ø,32,56,4,36,24,Ø
576 DATA Ø,28,32,56,36,36,24,Ø
578 DATA Ø,6Ø,4,8,16,32,32,Ø
58Ø DATA Ø,24,36,24,36,36,24,Ø
582 DATA Ø,24,36,36,28,4,56,Ø
584 DATA Ø,Ø,8,Ø,Ø,Ø,8,Ø
```

CHAPTER 2

Building Blocks, an Example Construction

In the previous chapter we got to grips with the fundamentals of BASIC programming. In this chapter we will not be concerning ourselves so much with understanding the programs but more with using them. We will work through an example of how to build up a game by adding subroutines to the control program. It may be helpful to think of the control program as being like a "blocks and holes" type of child's toy. To use it you must place one block in each hole to get a complete program. Remember that each block must be the right shape for the hole you are putting it in, although there may be variation in details (like colour) of the block. In the listing of the control program and in the other listings throughout this book, you will see many lines beginning with REM These lines are *not* instructions to the computer but are merely REMarks or REMinders to us humans of what we intended when we gave the instructions. If you wish to leave these lines out to save time typing it won't make any difference to the computer, the program or the game. It will mean, however, that you will need to take longer to find the section where some particular task is performed should you wish to examine or alter it later. Most people do not have perfect memories so these REMinders can be very important when trying to read the program.

Right then, let's make a game. Obviously the first thing we need to do is type, or LOAD from tape, the Control Program and the initialisation routines given in Chapter 1.

Having done this we need to decide what sort of game we wish to produce: What will be the aim of the game? Is it going to be a game of dodging ravenous spiders in the desert, or a game of blasting alien invaders out among the stars? Of course due to the nature of the system you could have cowboys shooting at alien flies in the middle of the ocean – whatever appeals to your sense of humour. It's entirely up to you. Well, whatever you decide the

first things to consider are whether you are going to shoot at things or dodge, and which modes of movement are available to your player. LEFT, RIGHT, UP, DOWN and even HYPER-space jumps can be selected. For our example game let's choose to be a tank holding off alien invaders amongst the stars.

Step 1. Choose the INSTRUCTIONS. Type in the first listing from the instructions section (listing 3.1a, also given below), which must always be used. Now we must select and add the lines that allow us to move LEFT and RIGHT (listings 3.1b and 3.1c). Finally we add the line which allows us to FIRE (listing 3.1f).

Listing 3.1a

```
1000 CLS:U=0:D=0:F=0:H=0:L=0:R=0
1010 PRINT@10,"INSTRUCTIONS"
1080 PRINT@451,"PRESS ANY KEY TO
 CONTINUE"
1090 A$=INKEY$:IF A$="" THEN 109
0
1095 RETURN
```

Listing 3.1b

```
1020 PRINT@130,"USE LEFT ARROW T
O MOVE LEFT":L=1
```

Listing 3.1c

```
1030 PRINT@162,"USE RIGHT ARROW
TO MOVE RIGHT":R=1
```

Listing 3.1f

```
1060 PRINT@258,"USE 0 TO FIRE":F
=1
```

Step 2. Choose the ALIEN. We have already decided that we want an alien invader so we'll use listing 3.2h.

Listing 3.2h

```
1100 REM ****************
1110 REM *ALIEN INVADER*
1120 REM ****************
1130 DATA 20,28,62,127,62,28,42,
73
1140 PG=2:N=1:GOSUB 9000
1150 REM ***********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ***********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

Step 3. Choose the PLAYER. We want a tank for our player so we'll use listing 3.3b. It is worth noting at this point that we could have chosen identical characters for both the alien and our player (e.g. both helicopters as in listings 3.2g and 3.3a) but it will look better if they are different.

Listing 3.3b

```
1200 REM **************
1210 REM *PLAYER TANK*
1220 REM **************
1230 DATA 16, 16,84, 124, 124, 124,6
8,0
1240 PG=2:N=1:GOSUB 9000
1250 REM ***********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM ***********************
1280 DIMB(1):GET(0, 144)-(7, 151),
B,G
1290 RETURN
```

Step 4. Choose the BACKGROUND. We could select any listing from section 3.4 of Chapter 3, but since we want an outer space game let's choose the scene with small stars (listing 3.4f).

Listing 3.4f

```
25 NV=Ø
13ØØ REM ****************
13Ø5 REM * SMALL STARS *
131Ø REM ****************
135Ø PMODE4,1:PCLSNV:SCREEN1,1
136Ø FOR Q=Ø TO 25
137Ø PSET(RND(255),RND(175),-(NV
=Ø))
138Ø NEXT Q
139Ø RETURN
```

Part of the general scene (or superimposed upon it) will be whatever we decide to display in the way of SCORES, AMMO and FUEL, so these headlines can be displayed here. We will ignore AMMO for the time being and just have FUEL and SCORE (listings 3.4i and 3.4j).

Listing 3.4i

```
1385 P$="SCORE:":XS=96:YS=176:GO
SUB 99ØØ
```

Listing 3.4j

```
1387 P$="FUEL:":XS=184:YS=176:GO
SUB 99ØØ
```

Step 5. START/RESTART routine. There is no choice about this one. We need to set all the variables even if we don't appear to be using them in this particular incarnation of our game. This is because some routines test them as a matter of course, and we

don't want AMMO=0 (for example) causing the game to end prematurely.

Listing 3.5

```
1400 REM **********************
1405 REM *SET UP AND RESTART*
1410 REM **********************
1420 ALIENS=10:REM *NO. OF ALIEN
S*
1430 SCORE=0:REM *SET SCORE TO Z
ERO*
1440 AMMO=10:REM *SET LASERS TO
FULL*
1450 FUEL=10:REM *SET FUEL TO FU
LL*
1460 PY=168:PX=120:REM *START PO
SITION OF SELF*
1470 IF F=0 THEN PY=8:REM *CHANG
E POSITION FOR DODGING GAME*
1475 GET(PX,PY)-(PX+7,PY+7),PB,G
:PUT(PX,PY)-(PX+7,PY+7),B,PSET
1480 DD=1:P=0:FIN=0
1490 XP=PX:YP=PY
1495 RETURN
```

Step 6. MOVE/FIRE routine. The main routine from this section must always be typed in, so first of all add listing 3.6a. Now we need the lines which allow us to move LEFT & RIGHT (listing 3.6b). Including this routine will only allow us to move LEFT or RIGHT provided we added the appropriate line in the INSTRUCTIONS routine.

Listing 3.6a

```
1500 REM ************
1502 REM *MOVE & FIRE*
1505 REM ************
```

```
1510 IF DD=0 THEN 1540
1520 DD=0
1530 AY=8:AX=RND(32)*8-8:XA=AX
1535 REM **MOVE ALIEN**
1540 IF AY<>8 THEN PUT(XA,AY-8)-
(XA+7,AY-1),AB,PSET
1545 GET(AX,AY)-(AX+7,AY+7),AB,G
:PUT(AX,AY)-(AX+7,AY+7),A,PSET
1550 XA=AX:AY=AY+8:IF AY=168 THE
N DD=1:P=P+1:ALIENS=ALIENS-1
1555 IF DD=1 THEN PUT(XA,AY-8)-(
XA+7,AY-1),AB,PSET
1560 AX=AX+8*(RND(3)-2)
1565 IF F=0 AND AX>PX THEN AX=AX
-8:GOTO 1580
1570 IF F=0 AND AX<PX THEN AX=AX
+8:GOTO 1580
1580 IF AX<0 THEN AX=0
1590 IF AX>248 THEN AX=248
1595 REM ****************
1596 REM * MOVE  PLAYER *
1599 REM ****************
1600 A$=INKEY$
1630 IF PX<0 THEN PX=0
1635 IF PY<0 THEN PY=0
1640 IF PX>248 THEN PX=248
1645 IF PY>168 THEN PY=168
1650 IF PY=YP AND PX=XP THEN 168
0
1660 PUT(XP,YP)-(XP+7,YP+7),PB,P
SET
1670 GET(PX,PY)-(PX+7,PY+7),PB,G
:PUT(PX,PY)-(PX+7,PY+7),B,PSET
1675 YP=PY:XP=PX
1680 K=0:IF F=1 AND A$="0" THEN
K=1
1690 RETURN
```

Listing 3.6b

```
1598 REM * LEFT & RIGHT *
1602 REM **LEFT AND RIGHT**
1605 IF L=1 AND A$=CHR$(8) THEN
PX=PX-8
1610 IF R=1 AND A$=CHR$(9) THEN
PX=PX+8
```

Step 7. CHECK FOR HIT routine. We must select a routine which will fire some sort of weapon so we won't use 3.7a. Let's use 3.7c, the LASER routine.

Listing 3.7c

```
1700 REM ***************
1701 REM *CHECK FOR HIT*
1702 REM *     LASER     *
1705 REM ***************
1710 HIT=0:IF K=0 THEN 1790
1715 SOUND100,1
1720 BX=PX+3:BY=167
1730 COLOR(1 AND NV=0),1:LINE(BX
,BY)-(BX,8),PSET
1745 IF BX>XA-1 AND BX<XA+8 THEN
 HIT=1
1755 COLORNV,1:LINE(BX,BY)-(BX,8
),PSET
1790 RETURN
```

Step 8. EXPLOSION routine. When we hit something we want to have some sort of acknowledgement of our success. We will choose a BLIP noise (listing 3.8i) for our LASER. We must add listing 3.8k because ours is a FIREing game and we need to wipe the alien from the screen when we hit it.

Listing 3.8i

```
1800 REM **************
1801 REM * EXPLOSION *
1802 REM *     BLIP    *
1805 REM **************
1810 FOR Q=1 TO 5
1820 SOUND150,1
1840 NEXT Q
1890 RETURN
```

Listing 3.8k

```
1880 IF F=1 THEN PUT(XA,AY-8)-(X
A+7,AY-1),AB,PSET:DD=1:ALIENS=AL
IENS-1
```

Step 9. SCORE routine. When we hit something we also want to have some sort of increase in our SCORE to tell us what great shots we are. We will choose to give ourselves 10 points for each alien we hit (listing 3.9d).

Listing 3.9d

```
1900 REM ******************
1901 REM *SCORING ROUTINE*
1902 REM *    10 POINTS    *
1905 REM ******************
1910 IF HIT=0 OR F=0 THEN 1990
1920 SCORE=SCORE+10
1990 RETURN
```

Step 10. FUEL & AMMUNITION routine. As we have decided to use FUEL and ignore AMMUNITION, we will start with listings 3.10b and 3.10f to decrease our FUEL, or energy reserves each time we fire our laser. We also want to increase the fuel when we score a hit and we will be generous and give ourselves a total reFUEL for each hit (listing 3.10h).

Listing 3.10b

```
2000 REM **********************
2002 REM *FUEL AND AMMUNITION*
2003 REM *                    *
2005 REM **********************
2090 RETURN
```

Listing 3.10f

```
2037 REM ****************
2038 REM *DECREASE FUEL*
2039 REM ****************
2040 IF K=1 THEN FUEL=FUEL-1
```

Listing 3.10h

```
2047 REM ****************
2048 REM * RESET FUEL  *
2049 REM ****************
2050 IF HIT=1 THEN FUEL=10
```

Step 11. STATUS DISPLAY. We need to update the display of information on the screen. For the SCORE we will need to use listing 3.11b. As we have already decided to tell the player how his fuel supply is progressing we will have to add listing 3.11d.

Listing 3.11b

```
2100 REM ******************
2102 REM *STATUS DISPLAY *
2103 REM *                *
2107 REM ******************
2110
2117 REM ****************
2118 REM *DISPLAY SCORE*
2119 REM ****************
```

```
2120 P$=STR$(SCORE)+"  ":XS=144:Y
S=176:GOSUB 9900
2190 RETURN
```

Listing 3.11d

```
2137 REM ***************
2138 REM *DISPLAY FUEL*
2139 REM ***************
2140 P$=STR$(FUEL)+"  ":XS=224:YS
=176:GOSUB 9900
```

Step 12. CHECK FOR END. Now we need to decide what conditions are going to indicate the end of the game. Obviously we will want to end the game at some time, so we start with listing 3.12a and add at least one of the other listings from that section. For our game let's choose to end the game if we let 3 of the opposition get PAST or if we run out of FUEL. So we have to add listings 3.12d and 3.12e.

Listing 3.12a

```
2200 REM ************************
2202 REM *CHECK FOR END OF GAME*
2205 REM ************************
2290 RETURN
```

Listing 3.12d

```
2240 IF P=3 THEN FIN=1
```

Listing 3.12e

```
2250 IF FUEL=0 THEN FIN=1
```

Step 13. END OF GAME DISPLAY. If it's the end of the game we need to either stop the game, or ask the player if he wants another go. Let's choose the POLITE STOP routine (listing 3.13b) and, since we have a SCORE, we'll also add listing 3.13c to tell us what we got.

Listing 3.13b

```
2300 REM **************
2302 REM * STOP GAME *
2303 REM *POLITE STOP*
2305 REM **************
2310 CLS
2330 PRINT@235,"ANOTHER GO?":A$=
INKEY$
2340 A$=INKEY$:IF A$="" THEN 234
0
2350 IF A$<>"N" THEN 2390
2360 STOP
2390 RETURN
```

Listing 3.13c

```
2320 PRINT@136,"YOUR SCORE WAS";
SCORE
```

And that's it! Your final listing should look like the example below. If it doesn't then make sure you've typed in all the listings with the right line numbers. Now all you have to do is key in RUN and press ENTER to play your game! GOOD SHOOTING!

If you have any problems or error messages then go back through your listing and check it corresponds exactly with the one below. The error message will tell you where the problem was noticed and it would be a good idea to look there first. Since the program is split up into sections you can check through just one section at a time. Of course the error might be in one of the previous sections and only have been noticed now. So, if you

can't see an error where it says, try working backwards. Remember, anything out of place or mis-typed will alter the program and could cause it to stop.

Example Program

```
10 REM ***************
12 REM *INITIALISATION*
14 REM ***************
20 PCLEAR6:PMODE4,3:PCLS
25 N=0
30 N=43:PG=3:GOSUB 9000
40 DIMPB(1):DIMN(1):DIMAB(1)
50 GM=0:REM *NO. OF GAMES PLAYED
*
100 REM **CONTROL PROGRAM**
102 '
104 REM **************
106 REM *INSTRUCTIONS*
110 GOSUB 1000
112 REM ****************
119 REM *GRAPHICS(ALIEN)*
120 GOSUB 1100
122 REM ****************
129 REM *GRAPHICS(SELF)*
130 GOSUB 1200
132 REM ************
139 REM *BACKGROUND*
140 GOSUB 1300:GM=GM+1
142 REM ****************
149 REM *START-UP/RESET*
150 GOSUB 1400
152 REM ***********
159 REM *MOVE/FIRE*
160 GOSUB 1500
162 REM ***************
169 REM *CHECK FOR HIT*
170 GOSUB 1700
```

```
172 REM **********
179 REM *EXPLOSION*
180 IF HIT=1 THEN GOSUB 1800
182 REM *********
189 REM *SCORING*
190 GOSUB 1900
192 REM ****************
199 REM *FUEL,LASERS ETC*
200 GOSUB 2000
202 REM ***************
209 REM *STATUS DISPLAY*
210 GOSUB 2100
212 REM **************
219 REM *END OF GAME?*
220 GOSUB 2200
222 REM ************
229 REM *ROUND AGAIN*
230 IF FIN=0 THEN 160
232 REM **********
239 REM *GAME OVER*
240 GOSUB 2300
242 REM ************
249 REM *START AGAIN*
250 GOTO 140
252 REM ************
500 DATA 0,60,66,66,126,66,66,0
502 DATA 0,124,66,124,66,66,124,
0
504 DATA 0,60,66,64,64,66,60,0
506 DATA 0,120,68,66,66,68,120,0
508 DATA 0,126,64,124,64,64,126,
0
510 DATA 0,126,64,124,64,64,64,0
512 DATA 0,60,66,64,78,66,60,0
514 DATA 0,66,66,126,66,66,66,0
516 DATA 0,62,8,8,8,8,62,0
518 DATA 0,2,2,2,66,66,60,0
520 DATA 0,68,72,112,72,68,66,0
```

```
522 DATA Ø,64,64,64,64,64,126,Ø
524 DATA Ø,66,102,90,66,66,66,Ø
526 DATA Ø,66,98,82,74,70,66,Ø
528 DATA Ø,60,66,66,66,66,60,Ø
530 DATA Ø,124,66,66,124,64,64,Ø
532 DATA Ø,60,66,66,114,74,60,Ø
534 DATA Ø,124,66,66,124,68,66,Ø
536 DATA Ø,60,64,60,2,66,60,Ø
538 DATA Ø,254,16,16,16,16,16,Ø
540 DATA Ø,66,66,66,66,66,60,Ø
542 DATA Ø,66,66,66,66,36,24,Ø
544 DATA Ø,66,66,66,90,102,66,Ø
546 DATA Ø,66,36,24,24,36,66,Ø
548 DATA Ø,130,68,40,16,16,16,Ø
550 DATA Ø,126,4,8,16,32,126,Ø
552 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
562 DATA 999,999,999,999,999
564 DATA Ø,24,36,44,52,36,24,Ø
566 DATA Ø,8,24,8,8,8,28,Ø
568 DATA Ø,24,36,8,16,32,60,Ø
570 DATA Ø,24,36,24,4,36,24,Ø
572 DATA Ø,8,24,40,72,124,8,Ø
574 DATA Ø,60,32,56,4,36,24,Ø
576 DATA Ø,28,32,56,36,36,24,Ø
578 DATA Ø,60,4,8,16,32,32,Ø
580 DATA Ø,24,36,24,36,36,24,Ø
582 DATA Ø,24,36,36,28,4,56,Ø
584 DATA Ø,Ø,8,Ø,Ø,Ø,8,Ø
999 REM instructions
1000 CLS:U=Ø:D=Ø:F=Ø:H=Ø:L=Ø:R=Ø
1010 PRINT@10,"INSTRUCTIONS"
1020 PRINT@130,"USE LEFT ARROW T
O MOVE LEFT":L=1
1030 PRINT@162,"USE RIGHT ARROW
TO MOVE RIGHT":R=1
1060 PRINT@258,"USE Ø TO FIRE":F
=1
```

```
1080 PRINT@451,"PRESS ANY KEY TO
 CONTINUE"
1090 A$=INKEY$:IF A$="" THEN 109
0
1095 RETURN
1099 REM alien graphics
1100 REM ****************
1110 REM *ALIEN INVADER*
1120 REM ****************
1130 DATA 20,28,62,127,62,28,42,
73
1140 PG=2:N=1:GOSUB 9000
1150 REM ********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
1199 REM self graphic
1200 REM **************
1210 REM *PLAYER TANK*
1220 REM **************
1230 DATA 16,16,84,124,124,124,6
8,0
1240 PG=2:N=1:GOSUB 9000
1250 REM ********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM ********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
1299 REM background
1300 REM ****************
1305 REM * SMALL STARS *
1310 REM ****************
1350 PMODE4,1:PCLSNV:SCREEN1,1
1360 FOR Q=0 TO 25
```

```
1370 PSET(RND(255),RND(175),-(NV
=0))
1380 NEXT Q
1385 P$="SCORE:":XS=96:YS=176:GO
SUB 9900
1387 P$="FUEL:":XS=184:YS=176:GO
SUB 9900
1390 RETURN
1400 REM **********************
1405 REM *SET UP AND RESTART*
1410 REM **********************
1420 ALIENS=10:REM *NO. OF ALIEN
S*
1430 SCORE=0:REM *SET SCORE TO Z
ERO*
1440 AMMO=10:REM *SET LASERS TO
FULL*
1450 FUEL=10:REM *SET FUEL TO FU
LL*
1460 PY=168:PX=120:REM *START PO
SITION OF SELF*
1470 IF F=0 THEN PY=8:REM *CHANG
E POSITION FOR DODGING GAME*
1475 GET(PX,PY)-(PX+7,PY+7),PB,G
:PUT(PX,PY)-(PX+7,PY+7),B,PSET
1480 DD=1:P=0:FIN=0
1490 XP=PX:YP=PY
1495 RETURN
1500 REM ************
1502 REM *MOVE & FIRE*
1505 REM ************
1510 IF DD=0 THEN 1540
1520 DD=0
1530 AY=8:AX=RND(32)*8-8:XA=AX
1535 REM **MOVE ALIEN**
1540 IF AY<>8 THEN PUT(XA,AY-8)-
(XA+7,AY-1),AB,PSET
```

```
1545 GET(AX,AY)-(AX+7,AY+7),AB,G
:PUT(AX,AY)-(AX+7,AY+7),A,PSET
1550 XA=AX:AY=AY+8:IF AY=168 THE
N DD=1:P=P+1:ALIENS=ALIENS-1
1555 IF DD=1 THEN PUT(XA,AY-8)-(
XA+7,AY-1),AB,PSET
1560 AX=AX+8*(RND(3)-2)
1565 IF F=0 AND AX>PX THEN AX=AX
-8:GOTO 1580
1570 IF F=0 AND AX<PX THEN AX=AX
+8:GOTO 1580
1580 IF AX<0 THEN AX=0
1590 IF AX>248 THEN AX=248
1595 REM **********************
1596 REM * MOVE  PLAYER *
1598 REM * LEFT & RIGHT *
1599 REM **********************
1600 A$=INKEY$
1602 REM **LEFT AND RIGHT**
1605 IF L=1 AND A$=CHR$(8) THEN
PX=PX-8
1610 IF R=1 AND A$=CHR$(9) THEN
PX=PX+8
1630 IF PX<0 THEN PX=0
1635 IF PY<0 THEN PY=0
1640 IF PX>248 THEN PX=248
1645 IF PY>168 THEN PY=168
1650 IF PY=YP AND PX=XP THEN 168
0
1660 PUT(XP,YP)-(XP+7,YP+7),PB,P
SET
1670 GET(PX,PY)-(PX+7,PY+7),PB,G
:PUT(PX,PY)-(PX+7,PY+7),B,PSET
1675 YP=PY:XP=PX
1680 K=0:IF F=1 AND A$="0" THEN
K=1
1690 RETURN
```

```
1700 REM ***************
1701 REM *CHECK FOR HIT*
1702 REM *    LASER    *
1705 REM ***************
1710 HIT=0:IF K=0 THEN 1790
1715 SOUND100,1
1720 BX=PX+3:BY=167
1730 COLOR(1 AND NV=0),1:LINE(BX
,BY)-(BX,8),PSET
1745 IF BX>XA-1 AND BX<XA+8 THEN
 HIT=1
1755 COLORNV,1:LINE(BX,BY)-(BX,8
),PSET
1790 RETURN
1800 REM **************
1801 REM * EXPLOSION *
1802 REM *   BLIP    *
1805 REM **************
1810 FOR Q=1 TO 5
1820 SOUND150,1
1840 NEXT Q
1880 IF F=1 THEN PUT(XA,AY-8)-(X
A+7,AY-1),AB,PSET:DD=1:ALIENS=AL
IENS-1
1890 RETURN
1900 REM ********************
1902 REM *SCORING 10 POINTS*
1905 REM ********************
1910 IF HIT=0 OR F=0 THEN 1990
1920 SCORE=SCORE+10
1990 RETURN
2000 REM ************************
2001 REM *                      *
2002 REM * FUEL AND AMMUNITION *
2003 REM *                      *
2004 REM ************************
```

```
2037 REM ******************
2038 REM * DECREASE FUEL *
2039 REM ******************
2040 IF K=1 THEN FUEL=FUEL-1
2047 REM ***************
2048 REM * RESET FUEL *
2049 REM ***************
2050 IF HIT =1 THEN FUEL=10
2090 RETURN
2100 REM ******************
2102 REM *STATUS DISPLAY *
2103 REM *                *
2107 REM ******************
2110 '
2117 REM ****************
2118 REM *DISPLAY SCORE*
2119 REM ****************
2120 P$=STR$(SCORE)+" ":XS=144:Y
S=176:GOSUB 9900
2137 REM ***************
2138 REM *DISPLAY FUEL*
2139 REM ***************
2140 P$=STR$(FUEL)+" ":XS=224:YS
=176:GOSUB 9900
2190 RETURN
2200 REM ************************
2202 REM *CHECK FOR END OF GAME*
2205 REM ************************
2240 IF P=3 THEN FIN=1
2250 IF FUEL=0 THEN FIN=1
2290 RETURN
2300 REM **************
2302 REM * STOP GAME *
2303 REM *POLITE STOP*
2305 REM **************
2310 CLS
2320 PRINT@136,"YOUR SCORE WAS";
SCORE
```

```
2330 PRINT@235,"ANOTHER GO?":A$=
INKEY$
2340 A$=INKEY$:IF A$="" THEN 234
0
2350 IF A$<>"N" THEN 2390
2360 STOP
2390 RETURN
9000 PMODE4,PG:SCREEN1,1:REMthis
 is just so you can see it happe
ning
9010 ST=7680+1536*(PG-2)
9020 FOR CH=0 TO N-1:RN=INT(CH/3
2)
9030 FOR Y=0 TO 7:READ CD:IF CD=
999 THEN Y=7:GOTO 9050
9035 IF NV=1 THEN CD=255-CD
9040 POKE ST+224*RN+CH+32*Y,CD
9050 NEXTY,CH
9055 RETURN
9899 REM *PRINT STRING*
9900 IF P$="" THEN RETURN
9910 A$=LEFT$(P$,1):P$=RIGHT$(P$
,LEN(P$)-1)
9920 IF A$=" " THEN YG=144:XG=20
8:GOTO 9950
9930 YG=144:AS=ASC(A$)-65:IF A$<
"A" THEN YG=152:AS=ASC(A$)-48
9940 XG=8*AS
9950 GOSUB 9960:XS=XS+8:GOTO 990
0
9960 PMODE4,3:GET(XG,YG)-(XG+7,Y
G+7),N,G
9970 PMODE4,1:PUT(XS,YS)-(XS+7,Y
S+7),N,PSET:RETURN
10000 A$="LISTING 3."
10010 INPUT"LISTING 3.WHAT";I$
10020 PRINT#-2,A$+I$
```

SAVING YOUR PROGRAM

When you are satisfied with your game you will probably want to SAVE it on cassette tape. This is done in the following way:

1. Position the tape in your cassette recorder, making sure that you have wound it past any plastic header tape as this cannot be recorded on.
2. Decide on a name for your program – you can use up to ten letters. For the purpose of this example let's use the name 'MY GAME'.
3. Now type in, without a line number:

    CSAVE "MY GAME"

4. Press the RECORD and PLAY keys together on your recorder then press ENTER. If you pressed ENTER before starting the tape you will either have to wait until the computer has finished and start again or you can stop it by pressing the RESET button on the left side of the computer. This will always stop the computer in its tracks and you will not lose your program. You will find that the BREAK keys will not interrupt a CSAVE (or CLOAD or SKIPF). However, be certain to press RESET and not the POWER SWITCH on the back of the computer. That certainly would cause the loss of your program.
5. Nothing much happens on the TV screen except that the flashing black cursor disappears. When the program has been saved the computer will once again chirp OK.
6. Now it would be nice to check that your program has been recorded properly. There is no foolproof way of doing this on the DRAGON but the following will usually fail with ?IO ERROR if there is anything wrong with the recording.

    First, rewind the tape and type either

    SKIPF "MY GAME"
    or
    SKIPF (on its own)

In either case press ENTER and start the tape (PLAY this time not RECORD!).

If it all goes fine and ends with OK then your program is probably recorded properly, but the only way you will ever know for sure is to switch off and on again and type CLOAD"MY GAME" followed by ENTER to load the program in again. I suggest that you have several successful SKIPFs before trying that.

If you have problems recording, try experimenting with different volume levels and, if your recorder has a tone control, turn it to maximum treble.

CHAPTER 3

Arcade Games, a Selection of Lego Bricks

This chapter is made up of thirteen sub-sections. Each of these sections contains a selection of routines which can be used to fill the holes in the control program. Remember to read the instructions at the start of each section which tells you whether listings are optional or not. If you don't feel too confident, why not follow the example routine and just use one different subroutine. Try, for example, a different background or a different alien and you will see how easy it is to make changes to the game. The more routines you decide to change the more your game will differ. Remember as long as you don't try to fit a square peg into a round hole you can use the routines in any combination. Some games may look a little bizarre, but at least they'll still be playable!

Following each section of listings is a brief explanation of how the routines in that section work. This includes details of what variables they change and how they relate to the other routines in the overall program.

We hope you enjoy experimenting with these routines as much as we enjoyed writing and testing them.

Section 1: INSTRUCTIONS

In this section 3.1a *always* needs to be used. This performs the basic tasks of getting ready to give instructions. Add to this any or all of the following listings 3.1b - g to complete the instructions block. It doesn't matter in what order you add the lines, because the computer will place them in numerical order in its listing.

Listing 3.1a

```
1000 CLS:U=0:D=0:F=0:H=0:L=0:R=0
1010 PRINT@10,"INSTRUCTIONS"
1080 PRINT@451,"PRESS ANY KEY TO
 CONTINUE"
1090 A$=INKEY$:IF A$="" THEN 109
0
1095 RETURN
```

This listing gives the instruction for, and enables, LEFT movement. You would normally include the RIGHT movement listing as well, but it's up to you.

Listing 3.1b

```
1020 PRINT@130,"USE LEFT ARROW T
O MOVE LEFT":L=1
```

This listing gives the instruction for, and enables, RIGHT movement.

Listing 3.1c

```
1030 PRINT@162,"USE RIGHT ARROW
TO MOVE RIGHT":R=1
```

This listing gives the instruction for, and enables, DOWNward movement. You would normally include the UPward movement listing as well, but again, it's up to you.

Listing 3.1d

```
1040 PRINT@194,"USE DOWN ARROW T
O MOVE DOWN":D=1
```

This listing gives the instruction for, and enables, UPward movement.

Listing 3.1e

```
1Ø5Ø PRINT@226,"USE UP ARROW TO
MOVE UP":U=1
```

This listing gives instructions for, and enables, FIREing.

Listing 3.1f

```
1Ø6Ø PRINT@258,"USE Ø TO FIRE":F
=1
```

This listing gives the instruction for, and enables, HYPER-drive for HYPER-space movement.

Listing 3.1g

```
1Ø7Ø PRINT@29Ø,"USE H TO HYPER-D
RIVE":H=1
```

The main instruction routine (listing 3.1a) clears the screen and PRINTs up a heading "INSTRUCTIONS". It also sets a number of variables that are used to indicate whether a particular ability is enabled. These variables (L,R,U,D,F and H) are all set to Ø (to indicate that they are disabled).

The other routines, when present, will be inserted in the list of instructions at this point. Each of these will PRINT a message on the screen telling you how to activate some action and also reset the appropriate flag variable to 1 (indicating that it is enabled). These flags are examined in other parts of the program to see whether or not the player is allowed to perform a particular action.

Flag variables are really just like any other variable, but we only give them the values 1 or 0. In this way they are used as indicators (either on or off), which can be quickly tested.

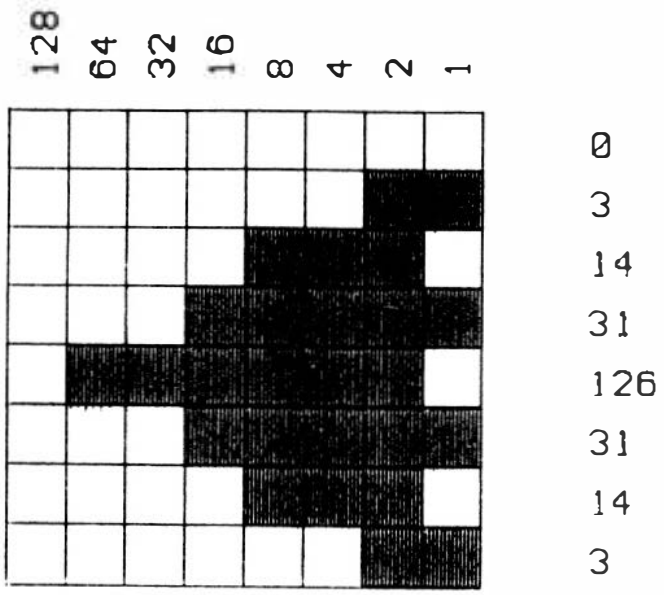
The final three lines of the main routine PRINT the message at the bottom of the screen and then wait until a key is depressed before RETURNing to the control program.

Section 2: ALIEN GRAPHICS

Choosing a shape for the aliens is very simple. Any of the following routines will define a character to use as the aliens. Each routine is accompanied by a display which shows the character's shape enlarged. It is important to remember at this stage that you will want to choose just one listing from this section and it can be any of the thirteen routines presented.

ALIEN FIGHTER

Fig. 3.2a

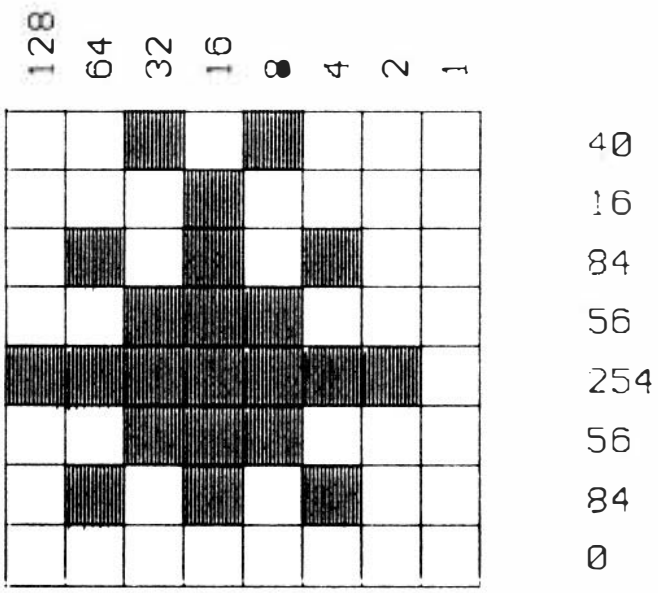


Listing 3.2a

```
1100 REM ****************
1110 REM *ALIEN FIGHTER*
1120 REM ****************
1130 DATA 0,3,14,31,126,31,14,3
1140 PG=2:N=1:GOSUB 9000
1150 REM ***********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ***********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN BUG

Fig. 3.2b



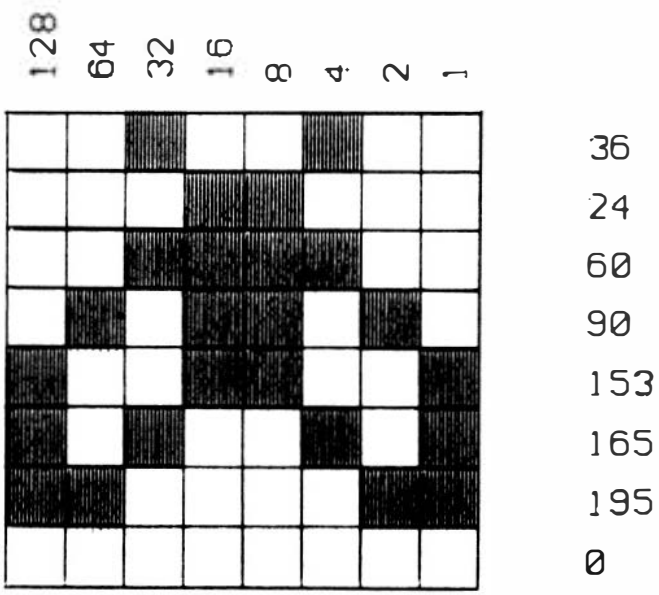
Listing 3.2b

```
1100 REM ************
1110 REM *ALIEN BUG*
1120 REM ************
```

```
1130 DATA 40,16,84,56,254,56,84,
0
1140 PG=2:N=1:GOSUB 9000
1150 REM **********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM **********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN FLY

Fig. 3.2c



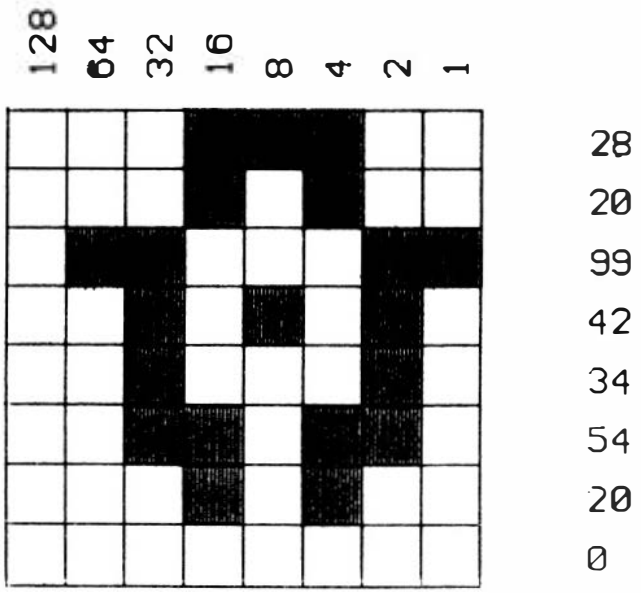
Listing 3.2c

```
1100 REM ***********
1110 REM *ALIEN FLY*
1120 REM ***********
1130 DATA 36,24,60,90,153,165,19
5,0
1140 PG=2:N=1:GOSUB 9000
```

```
1150 REM **********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM **********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN DESTROYER

Fig. 3.2d



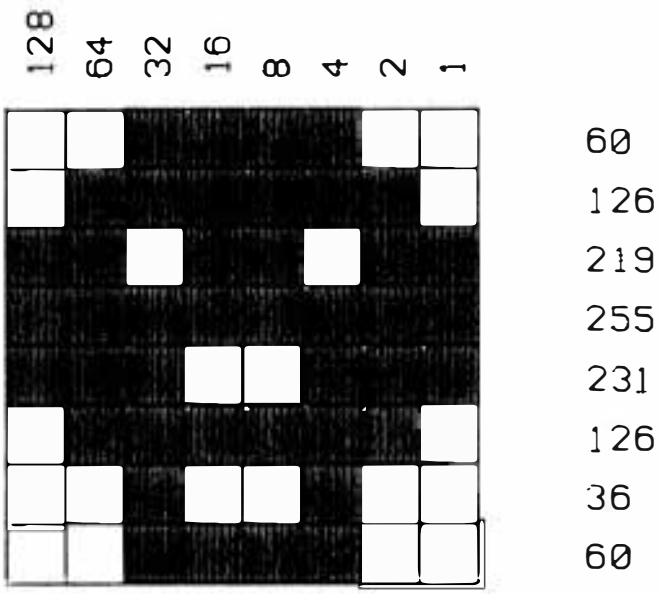
Listing 3.2d

```
1100 REM ******************
1110 REM *ALIEN DESTROYER*
1120 REM ******************
1130 DATA 28,20,99,42,34,54,20,0
1140 PG=2:N=1:GOSUB 9000
1150 REM **********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM **********************
```

```
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN SKULL

Fig. 3.2e

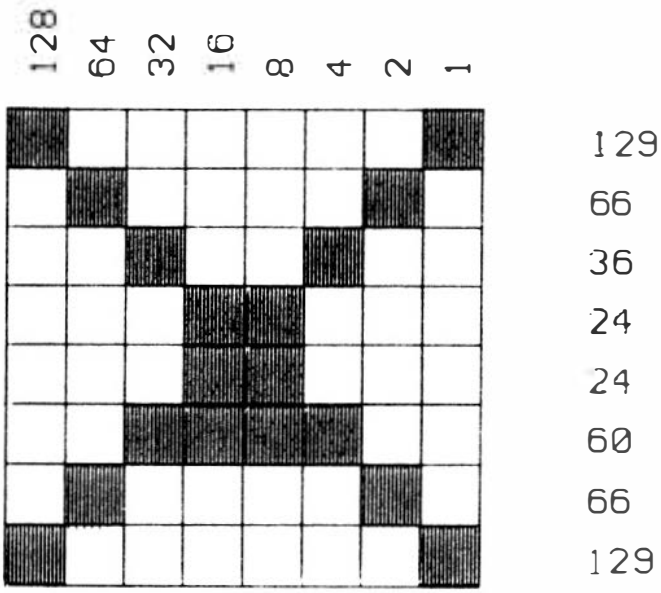


Listing 3.2e

```
1100 REM **************
1110 REM *ALIEN SKULL*
1120 REM **************
1130 DATA 60,126,219,255,231,126
,36,60
1140 PG=2:N=1:GOSUB 9000
1150 REM ***********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ***********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN INSECT

Fig. 3.2f

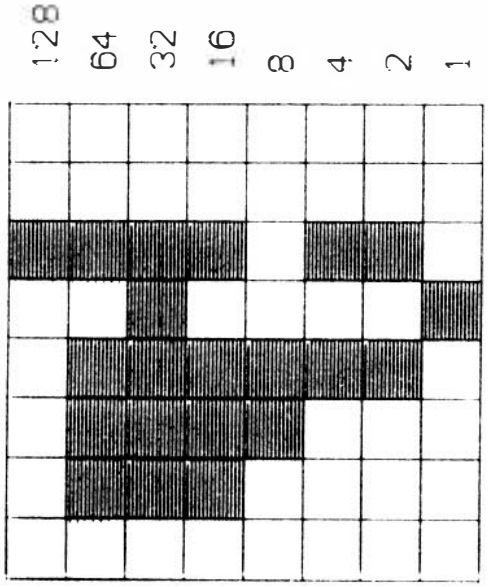


Listing 3.2f

```
1100 REM ***************
1110 REM *ALIEN INSECT*
1120 REM ***************
1130 DATA 129,66,36,24,24,60,66,
129
1140 PG=2:N=1:GOSUB 9000
1150 REM ***********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ***********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN HELICOPTER

Fig. 3.2g

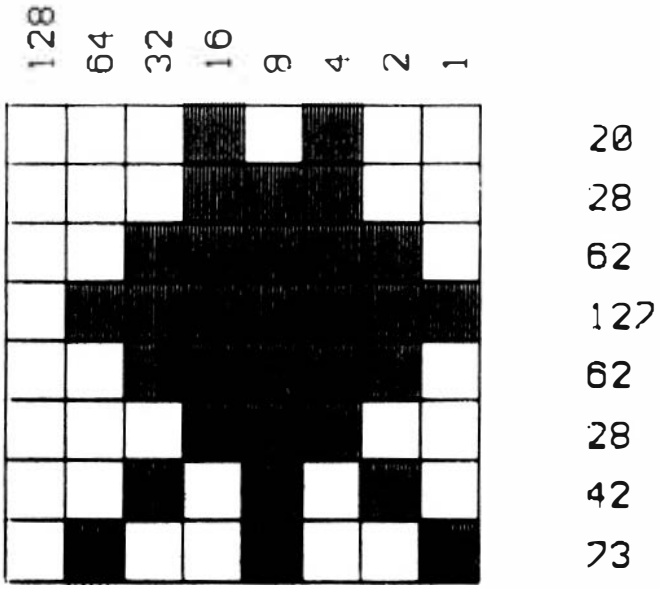


Listing 3.2g

```
1100 REM ********************
1110 REM *ALIEN HELICOPTER*
1120 REM ********************
1130 DATA 0,0,246,33,126,120,112
,0
1140 PG=2:N=1:GOSUB 9000
1150 REM ***********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ***********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN INVADER

Fig. 3.2h

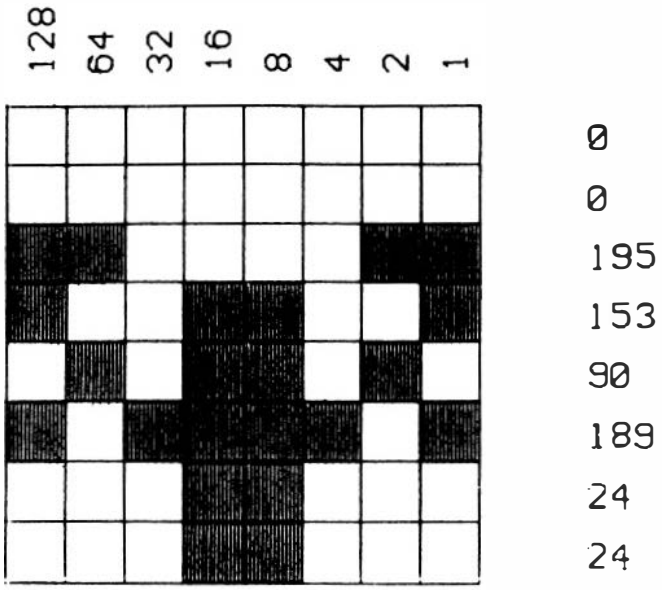


Listing 3.2h

```
1100 REM ****************
1110 REM *ALIEN INVADER*
1120 REM ****************
1130 DATA 20,28,62,127,62,28,42,
73
1140 PG=2:N=1:GOSUB 9000
1150 REM ***********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ***********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN BATTLESHIP

Fig. 3.2i

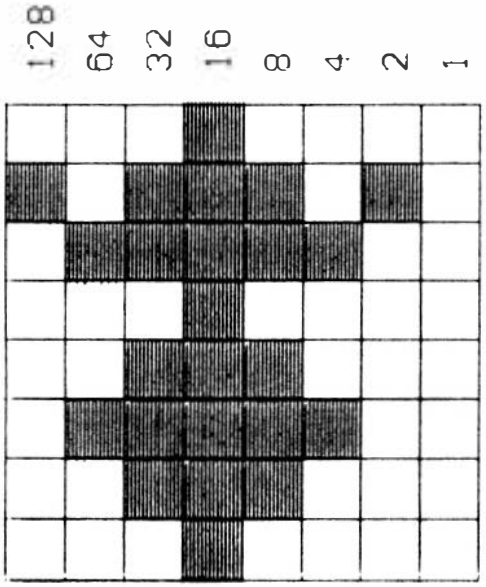


Listing 3.2i

```
1100 REM ********************
1110 REM *ALIEN BATTLESHIP*
1120 REM ********************
1130 DATA 0,0,195,153,90,189,24,
24
1140 PG=2:N=1:GOSUB 9000
1150 REM ***********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ***********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN CRUISER

Fig. 3.2j

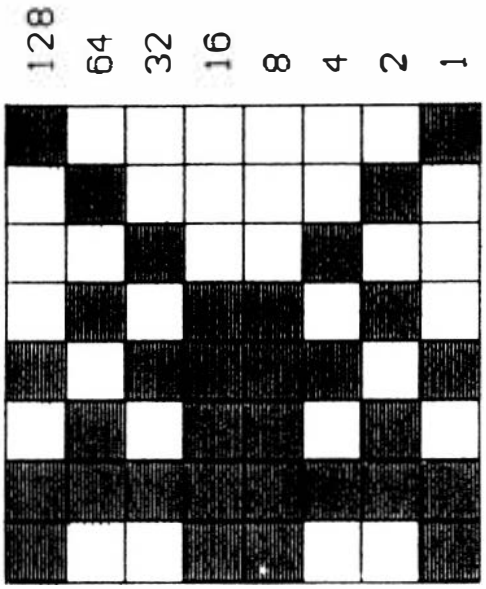


Listing 3.2j

```
1100 REM ****************
1110 REM *ALIEN CRUISER*
1120 REM ****************
1130 DATA 16,186,124,16,56,124,5
6,16
1140 PG=2:N=1:GOSUB 9000
1150 REM ********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN SPIDER

Fig. 3.2k

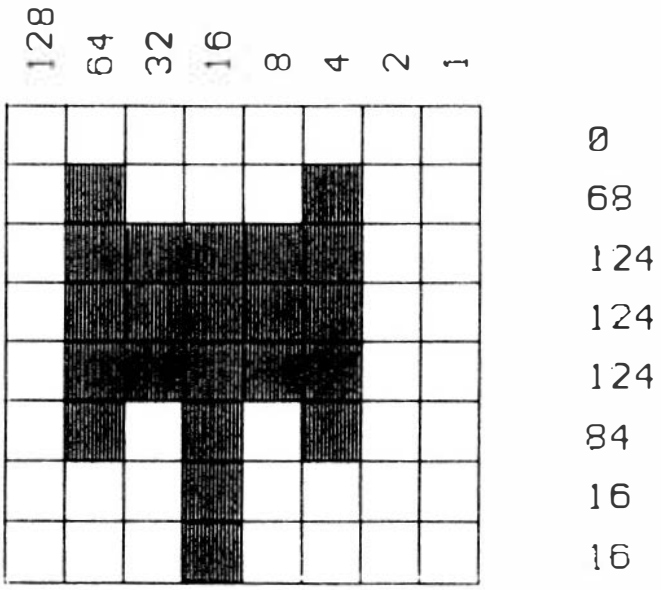


Listing 3.2k

```
1100 REM ***************
1110 REM *ALIEN SPIDER*
1120 REM ***************
1130 DATA 129,66,36,90,189,90,15
5,153
1140 PG=2:N=1:GOSUB 9000
1150 REM ********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN TANK

Fig. 3.21

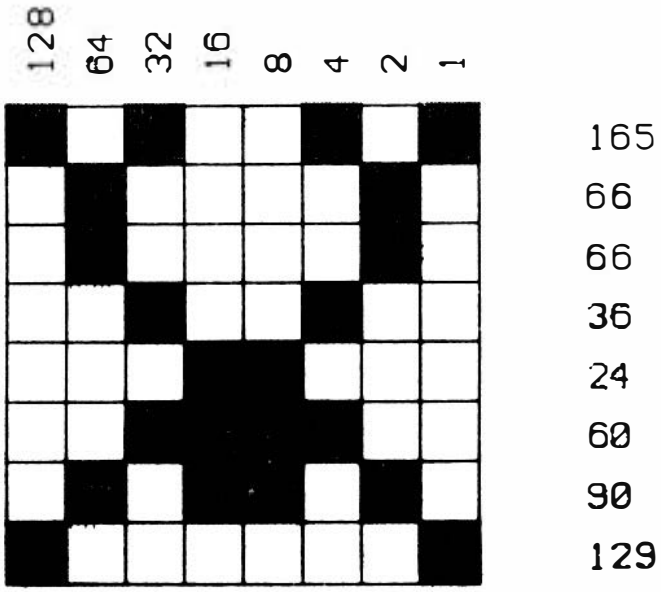


Listing 3.21

```
1100 REM ************
1110 REM *ALIEN TANK*
1120 REM ************
1130 DATA 0,68,124,124,124,84,16
,16
1140 PG=2:N=1:GOSUB 9000
1150 REM ********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

ALIEN FROG

Fig. 3.2m



Listing 3.2m

```
1100 REM ************
1110 REM *ALIEN FROG*
1120 REM ************
1130 DATA 165,66,66,36,24,60,90,
129
1140 PG=2:N=1:GOSUB 9000
1150 REM ********************
1160 REM *STORE IN ARRAY 'A'*
1170 REM ********************
1180 DIMA(1):GET(0,144)-(7,151),
A,G
1190 RETURN
```

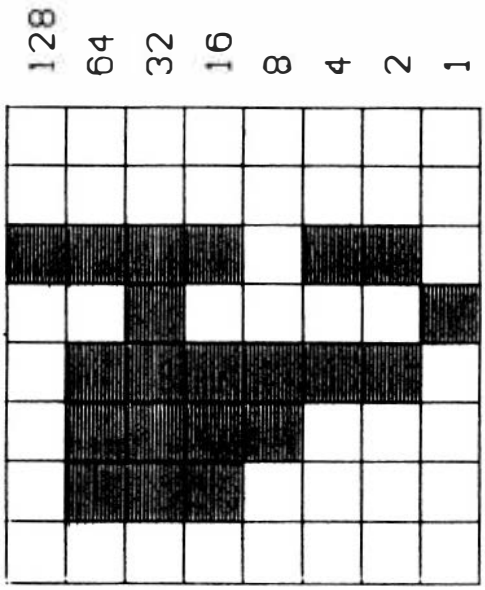
All the ALIEN graphics routines simply define a graphics character for the alien (for further explanation of this see Chapter 6). This character will be PUT on the screen at the alien's position.

Section 3: PLAYER GRAPHICS.

This is exactly the same as choosing a shape for the aliens. Any of the following routines will define a character for use as the player. Each routine is, once again, accompanied by a display to show the character's shape enlarged. Remember that you want the player to look different from the aliens. Some shapes might be more suitable than others depending on whether you are dodging or firing. You also need to take the background into account. Of course, it is possible to have a surfer blasting away with fireballs over a city skyline but it will look a bit strange. You need to choose just one listing from this section and it can be any of the fifteen routines presented.

PLAYER HELICOPTER

Fig. 3.3a



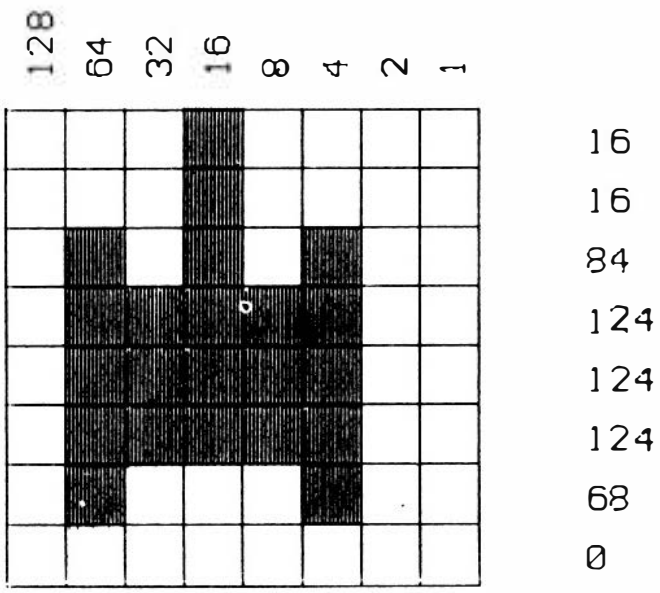
Listing 3.3a

```
12ØØ REM **********************
121Ø REM *PLAYER HELICOPTER*
122Ø REM **********************
123Ø DATA Ø,Ø,246,33,126,12Ø,112
,Ø
124Ø PG=2:N=1:GOSUB 9ØØØ
```

```
125Ø REM **********************
126Ø REM *STORE IN ARRAY 'B'*
127Ø REM **********************
128Ø DIMB(1):GET(Ø,144)-(7,151),
B,G
129Ø RETURN
```

PLAYER TANK

Fig. 3.3b

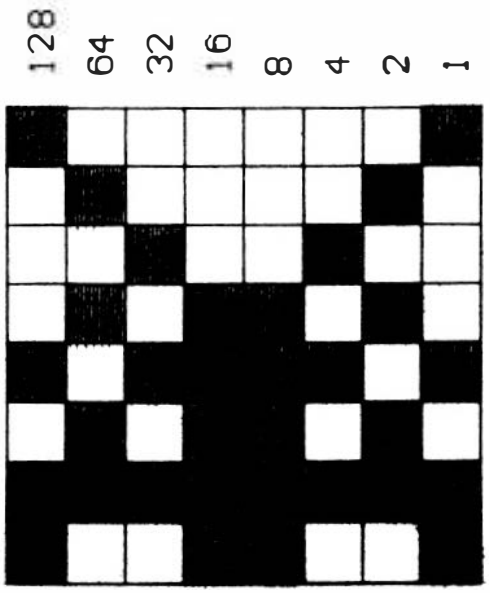


Listing 3.3b

```
12ØØ REM **************
121Ø REM *PLAYER TANK*
122Ø REM **************
123Ø DATA 16,16,84,124,124,124,6
8,Ø
124Ø PG=2:N=1:GOSUB 9ØØØ
125Ø REM **********************
126Ø REM *STORE IN ARRAY 'B'*
127Ø REM **********************
128Ø DIMB(1):GET(Ø,144)-(7,151),
B,G
129Ø RETURN
```

PLAYER SPIDER

Fig. 3.3c

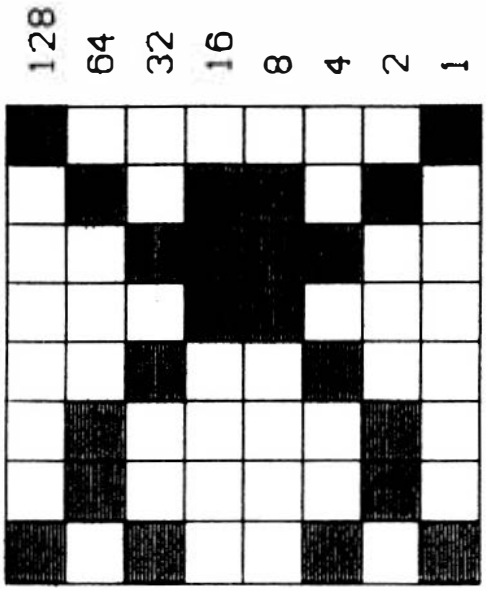


Listing 3.3c

```
1200 REM ****************
1210 REM *PLAYER SPIDER*
1220 REM ****************
1230 DATA 129,66,36,90,189,90,25
5,153
1240 PG=2:N=1:GOSUB 9000
1250 REM ***********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM ***********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

PLAYER FROG

Fig. 3.3d

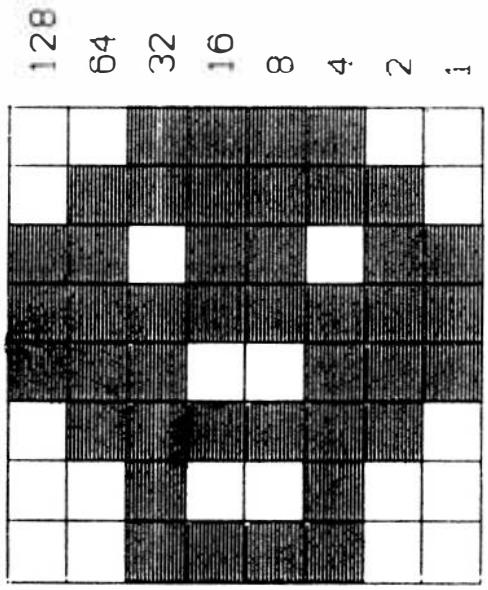


Listing 3.3d

```
1200 REM **************
1210 REM *PLAYER FROG*
1220 REM **************
1230 DATA 129,90,60,24,36,66,66,
165
1240 PG=2:N=1:GOSUB 9000
1250 REM ***********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM ***********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

PLAYER SKULL

Fig. 3.3e

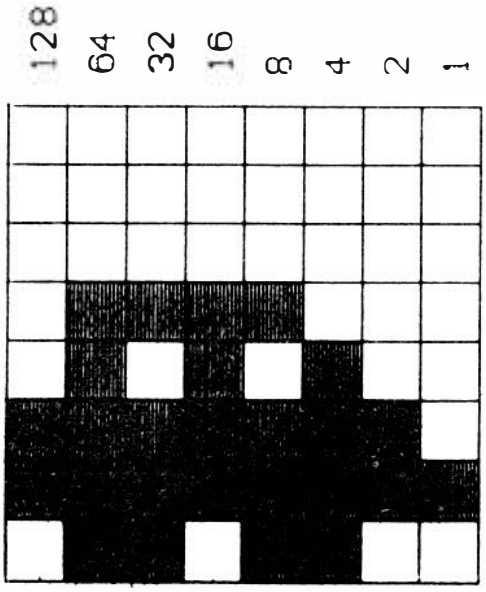


Listing 3.3e

```
1200 REM ***************
1210 REM *PLAYER SKULL*
1220 REM ***************
1230 DATA 60,126,219,255,231,126
,36,60
1240 PG=2:N=1:GOSUB 9000
1250 REM ***********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM ***********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

PLAYER AUTOMOBILE

Fig. 3.3f



Listing 3.3f

```
12ØØ REM **********************
121Ø REM *PLAYER AUTOMOBILE*
122Ø REM **********************
123Ø DATA Ø,Ø,Ø,12Ø,84,254,255,1
Ø8
124Ø PG=2:N=1:GOSUB 9ØØØ
125Ø REM **********************
126Ø REM *STORE IN ARRAY 'B'*
127Ø REM **********************
128Ø DIMB(1):GET(Ø,144)-(7,151),
B,G
129Ø RETURN
```

PLAYER AIRPLANE

Fig. 3.3g

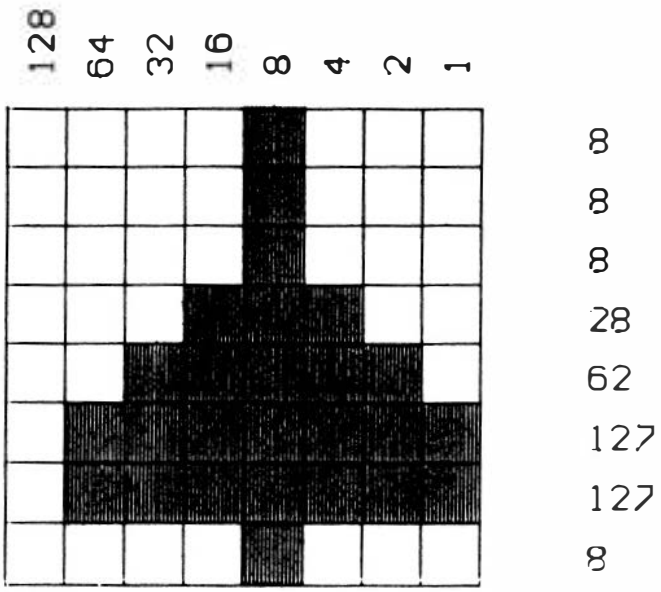
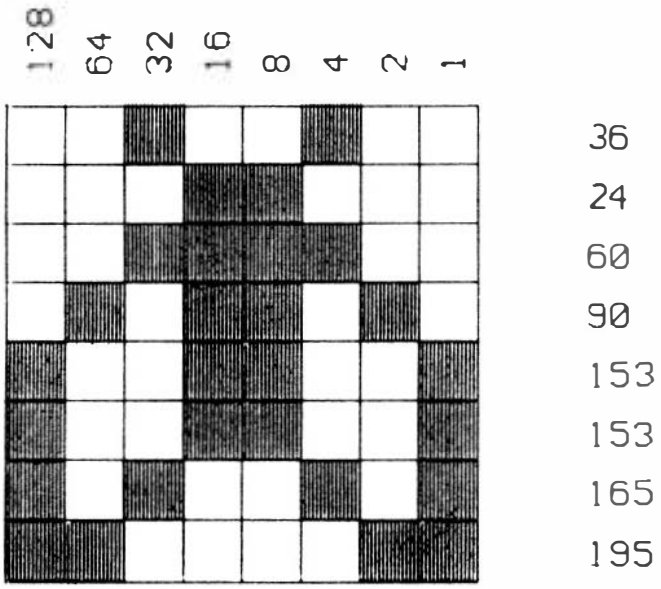


Fig. 3.3g

```
12ØØ REM ******************
121Ø REM *PLAYER AIRPLANE*
122Ø REM ******************
123Ø DATA 8,8,8,28,62,127,127,8
124Ø PG=2:N=1:GOSUB 9ØØØ
125Ø REM **********************
126Ø REM *STORE IN ARRAY 'B'*
127Ø REM **********************
128Ø DIMB(1):GET(Ø,144)-(7,151),
B,G
129Ø RETURN
```

PLAYER FLY

Fig. 3.3h

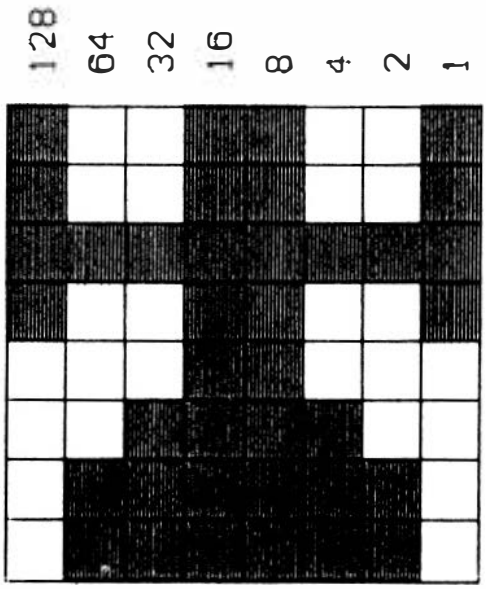


Listing 3.3h

```
1200 REM ************
1210 REM *PLAYER FLY*
1220 REM ************
1230 DATA 36,24,60,90,153,153,16
5,195
1240 PG=2:N=1:GOSUB 9000
1250 REM ********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM ********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

PLAYER BATTLESHIP

Fig. 3.3i

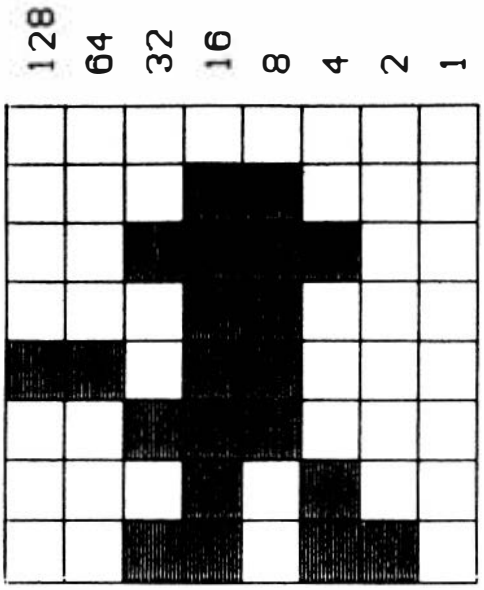


Listing 3.3i

```
1200 REM **********************
1210 REM *PLAYER BATTLESHIP*
1220 REM **********************
1230 DATA 153,153,255,153,24,60,
126,126
1240 PG=2:N=1:GOSUB 9000
1250 REM **********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM **********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

PLAYER COWBOY LEFT

Fig. 3.3j

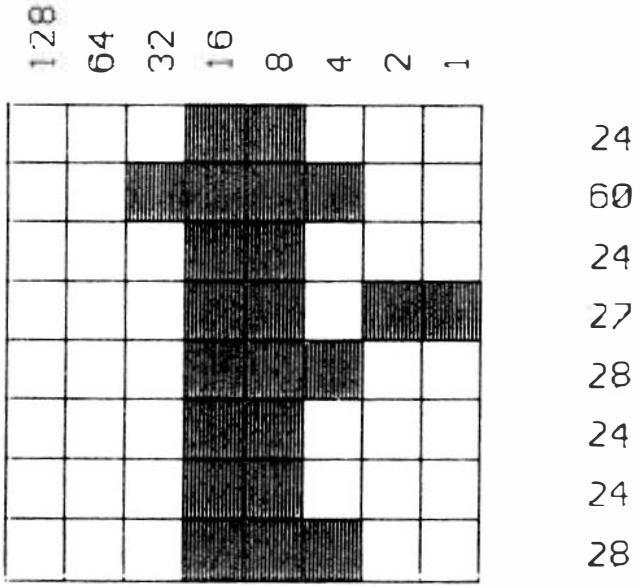


Listing 3.3j

```
1200 REM **********************
1210 REM *PLAYER COWBOY LEFT*
1220 REM **********************
1230 DATA 0,24,60,24,216,56,20,5
4
1240 PG=2:N=1:GOSUB 9000
1250 REM **********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM **********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

PLAYER COWBOY RIGHT

Fig. 3.3k

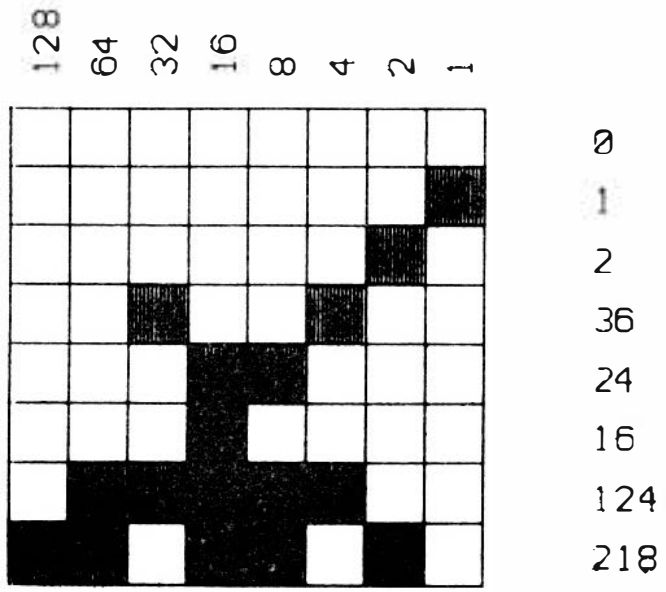


Listing 3.3k

```
1200 REM **********************
1210 REM *PLAYER COWBOY RIGHT*
1220 REM **********************
1230 DATA 24,60,24,27,28,24,24,2
8
1240 PG=2:N=1:GOSUB 9000
1250 REM **********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM **********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

PLAYER FIELD GUN

Fig. 3.31

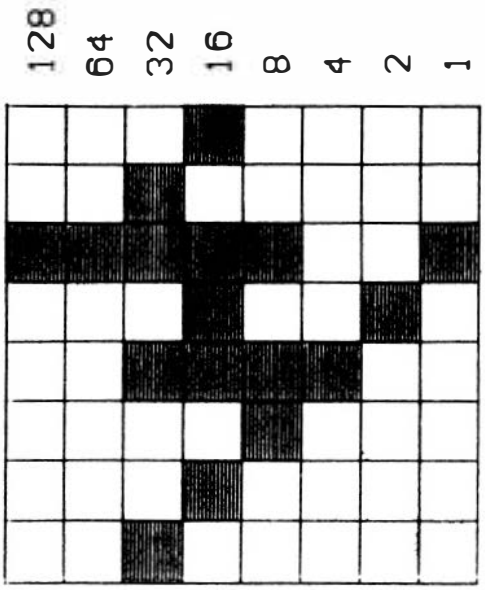


Listing 3.31

```
12ØØ REM ********************
1210 REM *PLAYER FIELD GUN*
1220 REM ********************
123Ø DATA Ø,1,2,36,24,16,124,218
124Ø PG=2:N=1:GOSUB 9ØØØ
125Ø REM **********************
126Ø REM *STORE IN ARRAY 'B'*
127Ø REM **********************
128Ø DIMB(1):GET(Ø,144)-(7,151),
B,G
129Ø RETURN
```

PLAYER SKIER

Fig. 3.3m

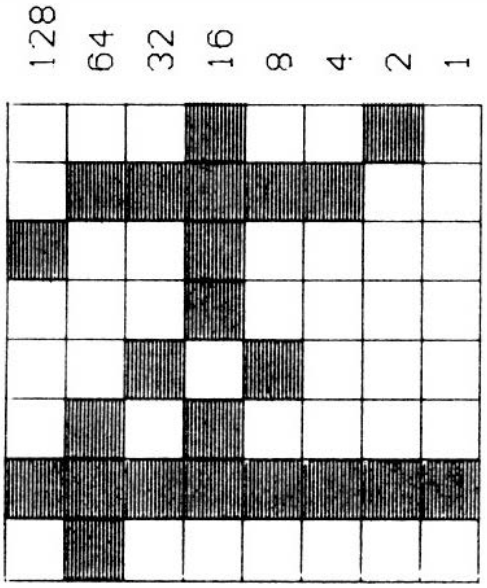


Listing 3.3m

```
1200 REM ***************
1210 REM *PLAYER SKIER*
1220 REM ***************
1230 DATA 16,32,249,18,60,8,16,3
2
1240 PG=2:N=1:GOSUB 9000
1250 REM ***********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM ***********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

PLAYER SURFER

Fig. 3.3n

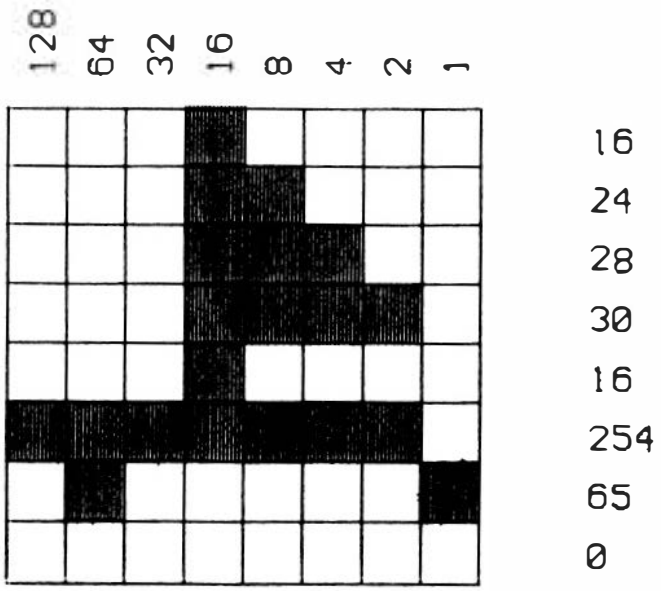


Listing 3.3n

```
1200 REM ****************
1210 REM *PLAYER SURFER*
1220 REM ****************
1230 DATA 18,124,144,16,40,80,25
5,64
1240 PG=2:N=1:GOSUB 9000
1250 REM ********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM ********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

PLAYER SAND YACHT

Fig. 3.3o



Listing 3.3o

```
1200 REM *********************
1210 REM *PLAYER SAND-YACHT*
1220 REM *********************
1230 DATA 16,24,28,30,16,254,65,
0
1240 PG=2:N=1:GOSUB 9000
1250 REM ***********************
1260 REM *STORE IN ARRAY 'B'*
1270 REM ***********************
1280 DIMB(1):GET(0,144)-(7,151),
B,G
1290 RETURN
```

All the player graphics routines simply define a character for the player (for further explanation of this see Chapter 6). This character will appear on the screen at the player's position.

Section 2: BACKGROUNDS

All the following routines set up a background scene against which your game will be played. This is fairly important, especially when it comes to dodging games. A skier switching back and forth through the pines whilst dodging spiders is going to have a harder time than one on a totally black background with no stationary objects to avoid. You need to choose just one of listings 3.4a to 3.4h, plus any of listings 3.4i, 3.4j and 3.4k according to whether or nor you want to display SCORE, AMMO and FUEL information. One more point to note is that each listing includes line 25 from the initialisation section of the program, which sets or resets the flag variable NV. This is an INVERSE flag and it swaps the foreground/background colours. These are normally white on black, but if we want a ski-slope, for example, then we obviously want the background to be white, not black. This flag also ensures that the characters are coloured the right way round. We will say more about this in Chapter 6. Each of the listings, apart from the first (totally black), is accompanied by a black and white representation of an example scene produced by that listing. This should help you decide which one to use.

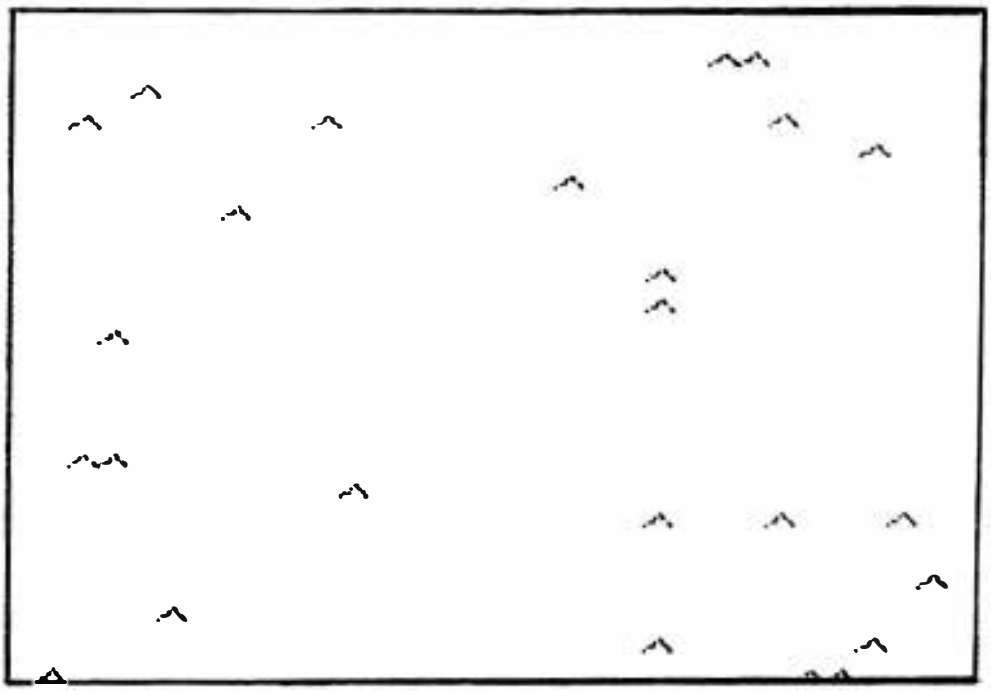
TOTALLY BLACK

Listing 3.4a

```
25 NV=Ø
13ØØ REM ******************
13Ø5 REM * TOTALLY BLACK *
131Ø REM ******************
135Ø PMODE4,1:PCLSNV:SCREEN1,1
139Ø RETURN
```

SEA-SCAPE

Fig. 3.4b

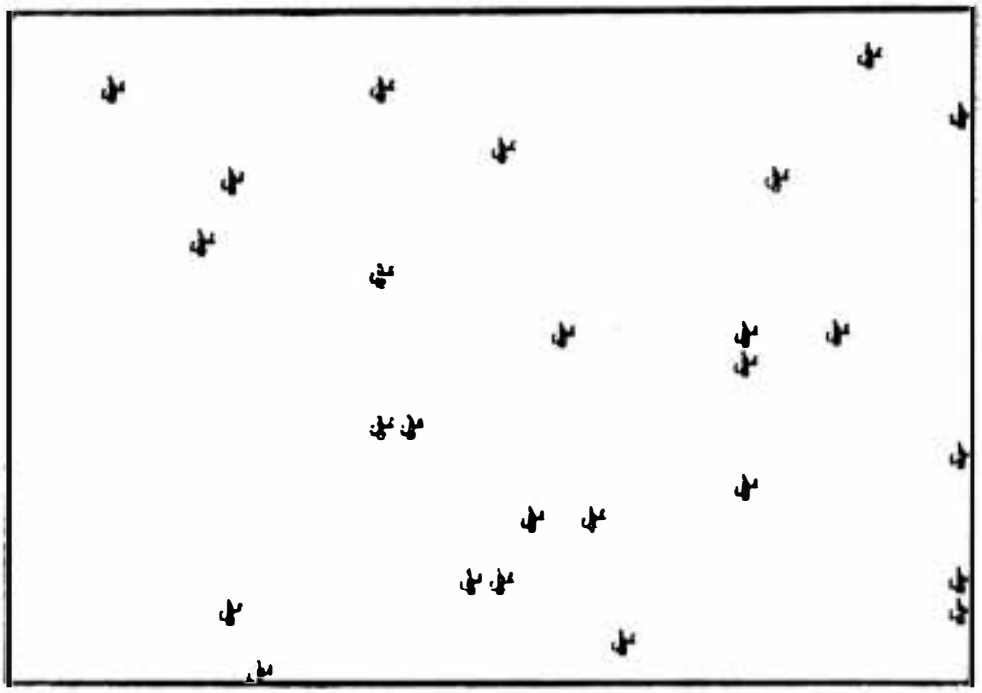


Listing 3.4b

```
25 NV=1
1300 REM **************
1305 REM * SEA-SCAPE *
1310 REM **************
1315 IF GM>0 THEN 1350
1320 PG=2:N=1:GOSUB 9000
1330 DATA 0,0,0,0,8,20,98,129
1340 DIMU(1):GET(0,144)-(7,151),
U,G
1350 PMODE4,1:PCLSNV:SCREEN1,0
1360 FOR Q=0 TO 25
1370 X2=RND(32)*8-8:Y2=RND(22)*8
-8:PUT(X2,Y2)-(X2+7,Y2+7),U,PSET
1380 NEXT Q
1390 RETURN
```

DESERT

Fig. 3.4c

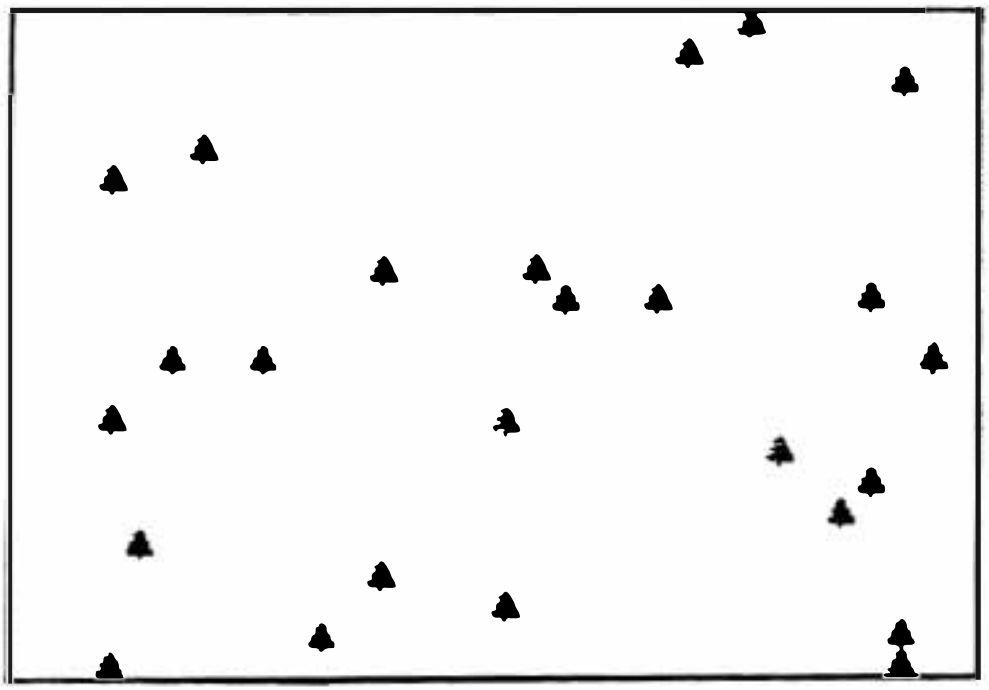


Listing 3.4c

```
25 NV=1
1300 REM **********
1305 REM * DESERT *
1310 REM **********
1315 IF GM>0 THEN 1350
1320 PG=2:N=1:GOSUB 9000
1330 DATA 0,16,18,26,94,88,120,2
4
1340 DIMU(1):GET(0,144)-(7,151),
U,G
1350 PMODE4,1:PCLSNV:SCREEN1,0
1360 FOR Q=0 TO 25
1370 X2=RND(32)*8-8:Y2=RND(22)*8
-8:PUT(X2,Y2)-(X2+7,Y2+7),U,PSET
1380 NEXT Q
1390 RETURN
```

SKI SLOPE

Fig. 3.4d

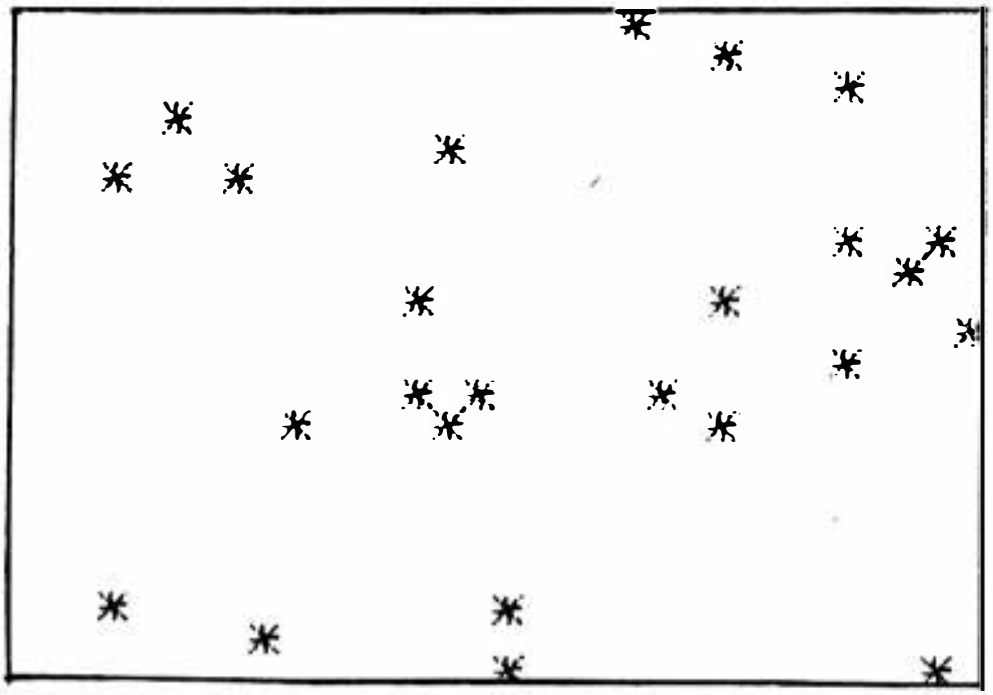


Listing 3.4d

```
25 NV=1
1300 REM **************
1305 REM * SKI-SLOPE *
1310 REM **************
1315 IF GM>0 THEN 1350
1320 PG=2:N=1:GOSUB 9000
1330 DATA 16,56,56,124,124,254,2
54,16
1340 DIMU(1):GET(0,144)-(7,151),
U,G
1350 PMODE4,1:PCLSNV:SCREEN1,1
1360 FOR Q=0 TO 25
1370 X2=RND(32)*8-8:Y2=RND(22)*8
-8:PUT(X2,Y2)-(X2+7,Y2+7),U,PSET
1380 NEXT Q
1390 RETURN
```

LARGE STARS

Fig. 3.4e

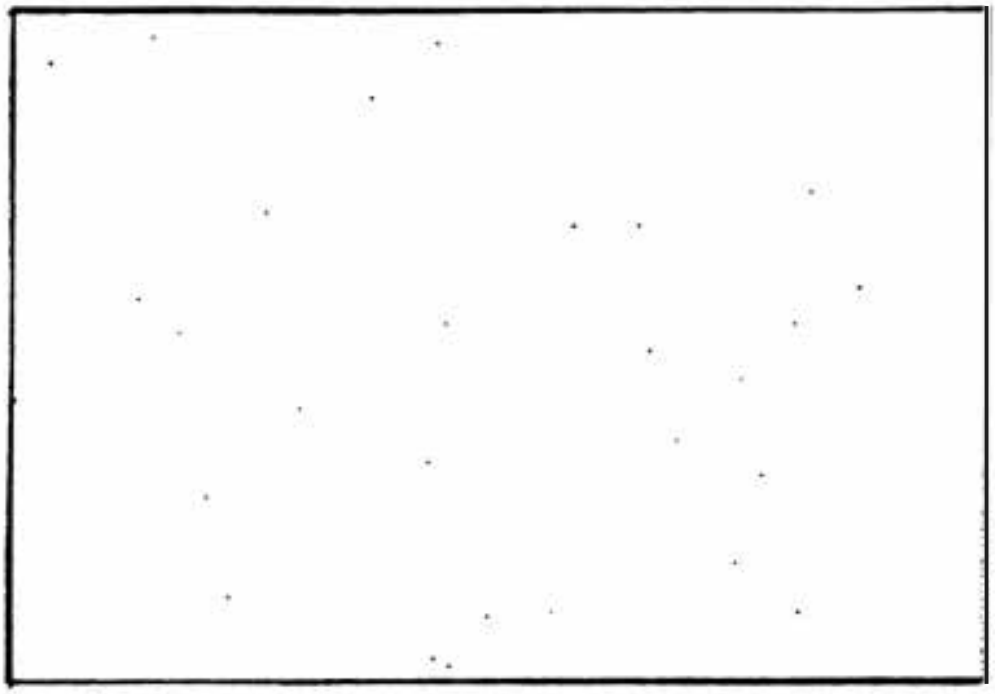


Listing 3.4e

```
25 NV=Ø
1300 REM ****************
1305 REM * LARGE STARS *
1310 REM ****************
1315 IF GM>Ø THEN 135Ø
1320 PG=2:N=1:GOSUB 9ØØØ
1330 DATA 145,82,52,31,248,44,74
,137
1340 DIMU(1):GET(Ø,144)-(7,151),
U,G
1350 PMODE4,1:PCLSNV:SCREEN1,1
1360 FOR Q=Ø TO 25
1370 X2=RND(32)*8-8:Y2=RND(22)*8
-8:PUT(X2,Y2)-(X2+7,Y2+7),U,PSET
1380 NEXT Q
1390 RETURN
```

SMALL STARS

Fig. 3.4f

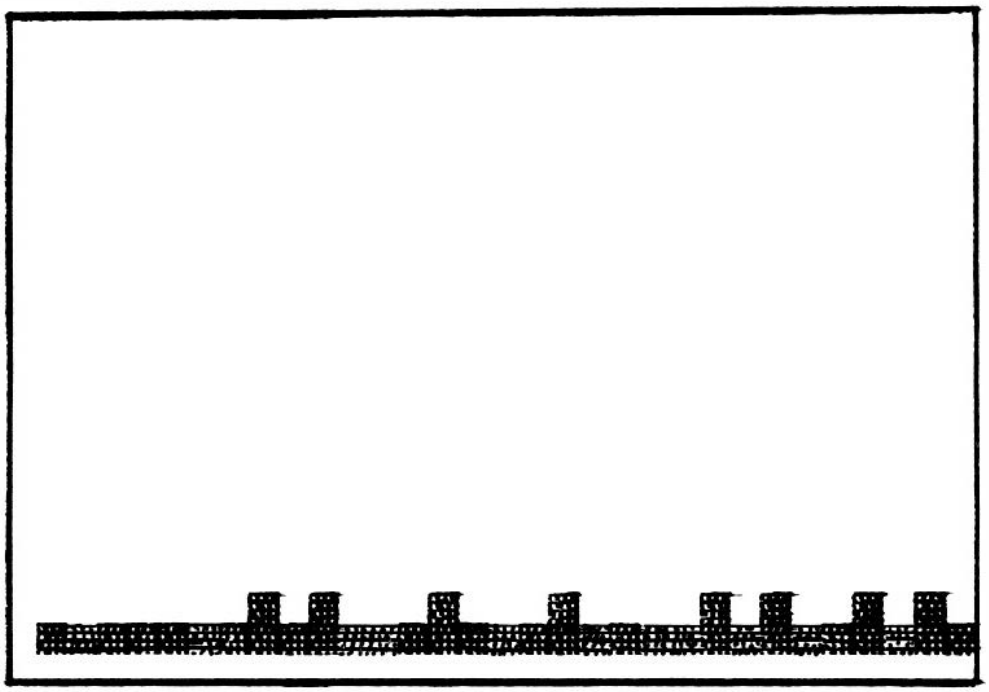


Listing 3.4f

```
25 NV=Ø
13ØØ REM ****************
13Ø5 REM * SMALL STARS *
131Ø REM ****************
135Ø PMODE4,1:PCLSNV:SCREEN1,1
136Ø FOR Q=Ø TO 25
137Ø PSET(RND(255),RND(175),-(NV
=Ø))
138Ø NEXT Q
139Ø RETURN
```

NIGHT SKYLINE

Fig. 3.4g

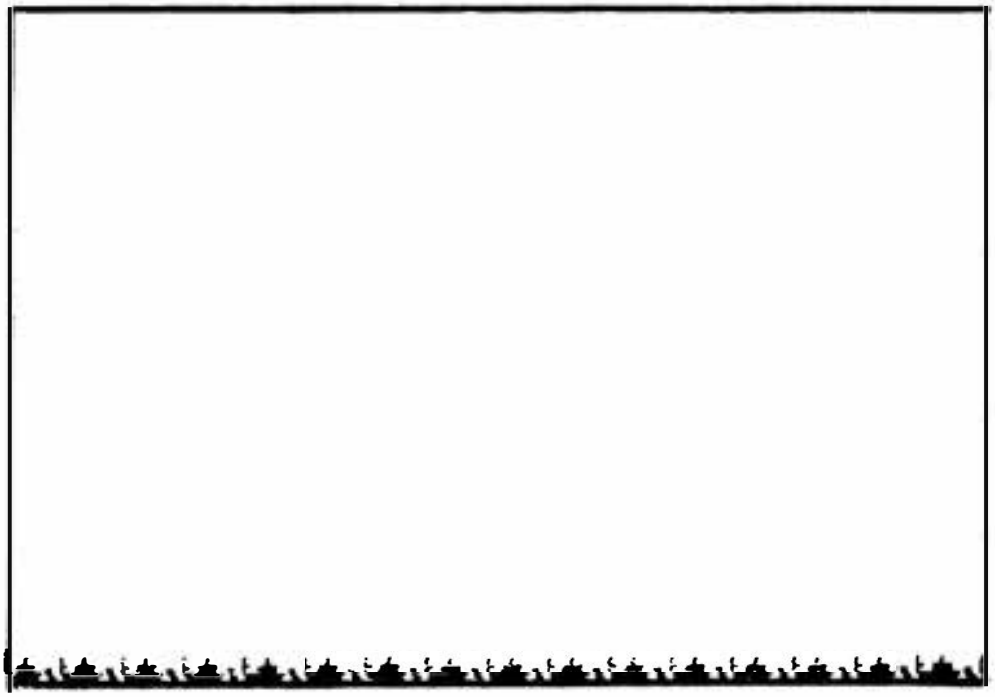


Listing 3.4g

```
25 NV=Ø
13ØØ REM ******************
13Ø5 REM * NIGHT SKYLINE *
131Ø REM ******************
1315 IF GM>Ø THEN 135Ø
132Ø PG=2:N=1:GOSUB 9ØØØ
133Ø DATA 255,171,255,181,255,21
3,255,181
134Ø DIMU(1):GET(Ø,144)-(7,151),
U,G
135Ø PMODE4,1:PCLSNV:SCREEN1,1
136Ø FOR Q=Ø TO 248 STEP 8
1365 FOR Y=Ø TO (RND(3)-1)*8 STE
P 8
137Ø PUT(Q,16Ø-Y)-(Q+7,167-Y),U,
PSET
138Ø NEXT Y:NEXT Q
139Ø RETURN
```

CITY SKYLINE

Fig. 3.4h



Listing 3.4h

```
25 NV=1
1300 REM ****************
1305 REM * CITY SKYLINE *
1310 REM ****************
1315 IF GM>0 THEN 1350
1320 PG=2:N=2:GOSUB 9000
1330 DATA 2,2,3,50,179,25,255,25
5,8,8,62,62,255,255,255,255
1340 DIMU(1):GET(0,144)-(7,151),
U,G
1345 DIMT(1):GET(8,144)-(15,151)
,T,G
1350 PMODE4,1:PCLSNV:SCREEN1,1
1360 FOR Q=0 TO 248 STEP 16
1370 PUT(Q,160)-(Q+7,167),U,PSET
```

```
1375 PUT(Q+8,160)-(Q+15,167),T,P
SET
1380 NEXT Q
1390 RETURN
```

The background routines select PMODE4 with graphics pages 1 to 4 on screen. They then clear the screen to the background colour (which depends on the setting of the NV flag). In the first routine, listing 3.4a, this is all that is required. For the rest of the routines a character for a suitable object is defined (for further explanation of this see Chapter 6). This object is then PUT on the screen in various random positions. (see Chapter 10). Listing 3.4f differs from this general pattern by merely PSETing points at these random positions instead of PUTting a character each time. Listings 3.4g and 3.4h do use characters but use special methods of determining the positions where these will be PUT to achieve the desired effect in each case. These routines are examined separately and in greater detail, in Chapter 4.

The last three listings PUT the status headings on the screen by way of the PUTSTRING routine – this will be more fully explained in Chapter 5.

Section 5: START/RESTART.

This routine is *not* optional. We *always* need to have this routine exactly as listed. This ensures that all the variables which might be used later on are initialised whether our game takes any account of them or not.

Listing 3.5

```
1400 REM **********************
1405 REM *SET UP AND RESTART*
1410 REM **********************
```

```
1420 ALIENS=10:REM *NO. OF ALIEN
S*
1430 SCORE=0:REM *SET SCORE TO Z
ERO*
1440 AMMO=10:REM *SET LASERS TO
FULL*
1450 FUEL=10:REM *SET FUEL TO FU
LL*
1460 PY=168:PX=120:REM *START PO
SITION OF PLAYER*
1470 IF F=0 THEN PY=8:REM *CHANG
E POSITION FOR DODGING GAME*
1475 GET(PX,PY)-(PX+7,PY+7),PB,G
:PUT(PX,PY)-(PX+7,PY+7),B,PSET
1480 DD=1:P=0:FIN=0
1490 XP=PX:YP=PY
1495 RETURN
```

This routine essentially simply initialises all the variables used in the game. For greater detail of what the variables are actually used for see Appendix One. The number of ALIENS is set to 10, your SCORE is set to zero and you are given 10 units of AMMO and FUEL. These variables must always be set to some starting value even if they're not used by the game.

PX and PY are the horizontal (x) and vertical (y) co-ordinates on screen of the player's character and these must be set to some initial values. For firing games you start off at the bottom of the screen with PY=168 (we leave some space at the bottom for scores, etc.), and with PX=120 which is about halfway across. If the game is not a firing game you start at the top of the screen instead, with PY=8. Your character is now PUT on to the screen at its initial position, but first the bit of scenery at the position is stored in the array PB (for player background) with the GET instruction. This is so that we can PUT the scenery back again when you move. To do this we need to know where the player was before he moved, so we store a copy of the player's old position in the variables XP and YP (the reverse of PX and PY).

Finally we set the flags that indicate whether a new alien is required and whether the end of the game has arrived. To do this we set DD (for dead) to 1 and set FIN to 0. (The French is because END is a special BASIC word for the DRAGON computer, so we cannot use it as a variable.) We also need to reset the counter which tells us how many aliens have got past, so we set P equal to 0.

Section 6: MOVE/FIRE.

We must have the first, main routine from this section in our program. If you allowed LEFT & RIGHT movement then you will need to add listing 3.6b. Similarly listing 3.6c must be added for UP & DOWN movement. Don't worry if you have only allowed, say, LEFT movement in the instructions. Just add the section for both LEFT & RIGHT and you will find that only LEFT movement is actually enabled. If you have chosen HYPER-drive as a mode of movement than you will need to add listing 3.6d.

Listing 3.6a

```
1500 REM **************
1502 REM *MOVE & FIRE*
1505 REM **************
1510 IF DD=0 THEN 1540
1520 DD=0
1530 AY=8:AX=RND(32)*8-8:XA=AX
1535 REM **MOVE ALIEN**
1540 IF AY<>8 THEN PUT(XA,AY-8)-
(XA+7,AY-1),AB,PSET
1545 GET(AX,AY)-(AX+7,AY+7),AB,G
:PUT(AX,AY)-(AX+7,AY+7),A,PSET
1550 XA=AX:AY=AY+8:IF AY=168 THE
N DD=1:P=P+1:ALIENS=ALIENS-1
1555 IF DD=1 THEN PUT(XA,AY-8)-(
XA+7,AY-1),AB,PSET
```

```
1560 AX=AX+8*(RND(3)-2)
1565 IF F=0 AND AX>PX THEN AX=AX
-8:GOTO 1580
1570 IF F=0 AND AX<PX THEN AX=AX
+8:GOTO 1580
1580 IF AX<0 THEN AX=0
1590 IF AX>248 THEN AX=248
1595 REM ****************
1596 REM * MOVE  PLAYER *
1599 REM ****************
1600 A$=INKEY$
1630 IF PX<0 THEN PX=0
1635 IF PY<0 THEN PY=0
1640 IF PX>248 THEN PX=248
1645 IF PY>168 THEN PY=168
1650 IF PY=YP AND PX=XP THEN 168
0
1660 PUT(XP,YP)-(XP+7,YP+7),PB,P
SET
1670 GET(PX,PY)-(PX+7,PY+7),PB,G
:PUT(PX,PY)-(PX+7,PY+7),B,PSET
1675 YP=PY:XP=PX
1680 K=0:IF F=1 AND A$="0" THEN
K=1
1690 RETURN
```

The routine below allows LEFT or RIGHT movement providing that the INSTRUCTIONS enable such movement.

Listing 3.6b

```
1598 REM * LEFT & RIGHT *
1602 REM **LEFT AND RIGHT**
1605 IF L=1 AND A$=CHR$(8) THEN
PX=PX-8
1610 IF R=1 AND A$=CHR$(9) THEN
PX=PX+8
```

This routine allows UP or DOWN movement providing that the INSTRUCTIONS enable such movement.

Listing 3.6c

```
1597 REM *  UP & DOWN   *
1612 REM **UP AND DOWN**
1615 IF U=1 AND A$=CHR$(94) THEN
 PY=PY-8
1620 IF D=1 AND A$=CHR$(10) THEN
 PY=PY+8
```

This next routine allows HYPER-space movement providing that the INSTRUCTIONS enable such movement.

Listing 3.6d

```
1622 REM **HYPER JUMPS**
1625 IF H=1 AND A$="H" THEN PX=8
*RND(31):PY=8*RND(21)
```

The routine to control movement and firing is quite long and performs several different tasks, so let's look at them in order. Firstly it checks whether a new alien is required (if the previous one is dead – DD=1 – then we do need one). If we do start a new alien then we must initialise the horizontal (x) and vertical (y) screen co-ordinates for the alien. We set RY equal to 8 and RX equal to a random multiple of 8 between 8 and 248. As with the player we will need to remember the position of the alien so we copy AX into XA. We do not need to remember the y co-ordinate because it will always 8 less than the current y co-ordinate. Now we set the flag to show that the alien is not dead (DD=0).

By now we have a live alien on our hands (well, on the screen anyway) so we want to move it. We first PUT back the background (array AB) over the alien to erase him from his old

position, and then we GET the background for his new position, before PUTting the alien at his new position. We must now calculate the next position for our alien. We add 8 to AY to move him down the screen. At this point we test whether he has reached the bottom of the screen (AY=168). If he has, then he got past us, so we add one to P (the past variable), and declare that alien dead (DD=1). We also take one off the number of aliens left (ALIENS=ALIENS-1). If the alien is dead then we PUT back the scenery on top of him to erase him (or bury him).

Assuming he's not dead we would like our alien to move from side to side as he descends. If it's a dodging game (not a firing game, so F=0) we want the alien to move towards us so we test which side of your x position (PX) he is and alter AX accordingly.

If he is directly above you, or if it a firing game, then we will alter his horizontal position by one either side at random. Having changed the alien's position we check that he hasn't gone off the side of the screen and reset AX if he has. (Some funny things happen if you try to PUT off the screen.)

Now we come to moving the player. It is at this stage that we will encounter some of the lines from the other listings when they have been added. If a particular sort of movement is allowed we check to see if the appropriate key is being pressed and then alter PX or PY accordingly. If HYPER-space is included and the H key has been pressed then we choose new values for PX and PY at random (again in multiples of 8). Once again we must check that the new co-ordinates are still on the screen and reset them if they are not. Finally we PUT back the background (array PB) at the player's old position, GET the background at his new position, and PUT the player at the new position. Note that we do not do this if the player has not moved since the last time around. This is to prevent the player graphic from flickering when stationary.

The final action in this section is to check the FIRE button. If this key has been pressed then we set the flag K to 1 to indicate that a shot is to be fired.

Section 7: DETECT A HIT

This routine can have two completely different forms. This depends on whether we are playing a firing game or a dodging game. In the latter case what we need to detect are collisions between the player and other objects. These objects may be background objects or aliens. The first routine (listing 3.7a) in this section deals with the detection of this sort of collision and should be used if yours is a dodging game.

If we are playing a firing game then we might need to move a missile up the screen and see if this hits any alien. For this type of game we can pick either of the other two routines in this section (listings 3.7b or 3.7c). These shoot bullets and lasers respectively. Later, in Chapter 6, we will add another two routines to our selection which shoot bombs and fireballs, but these will have to wait until we have some techniques under our belts.

Listing 3.7a

```
1700 REM ***************
1701 REM *CHECK FOR HIT*
1702 REM * DODGING GAME*
1705 REM ***************
1710 HIT=0
1720 PMODE4,2:N=255*NV
1730 PUT(0,144)-(7,151),PB,PSET
1740 FOR Y=7680 TO 7904 STEP 32
1750 IF PEEK(Y)<>N THEN HIT=1
1760 NEXT Y
1770 PMODE4,1
1790 RETURN
```

Listing 3.7b

```
1700 REM ***************
1701 REM *CHECK FOR HIT*
1702 REM *    BULLET     *
1705 REM ***************
```

```
1710 HIT=0:IF K=0 THEN 1790
1715 SOUND100,1
1720 BX=PX+3:BY=167
1725 PSET(BX,BY,-(NV=0))
1730 BY=BY-4
1735 PSET(BX,BY,-(NV=0))
1740 PSET(BX,BY+4,NV)
1745 IF XA=PX AND AY/8=INT(BY/8)
 THEN HIT=1:GOTO 1755
1750 IF BY>11 THEN 1730
1755 PSET(BX,BY,NV)
1790 RETURN
```

Listing 3.7c

```
1700 REM ****************
1701 REM *CHECK FOE HIT*
1702 REM *     LASER     *
1705 REM ****************
1710 HIT=0:IF K=0 THEN 1790
1715 SOUND100,1
1720 BX=PX+3:BY=167
1730 COLOR(1 AND NV=0),1:LINE(BX
,BY)-(BX,8),PSET
1745 IF BX>XA-1 AND BX<XA+8 THEN
 HIT=1
1755 COLORNV,1:LINE(BX,BY)-(BX,8
),PSET
1790 RETURN
```

The first routine in this section works by examining the background underneath the player. This is stored in the array PB. There is no easy direct way of examining an array which has been used with GET, so what we do is to PUT the array onto the hidden screen (PAGE 6) and then examine that part of the screen for any foreground colour. If we find any then obviously the player has hit something.

The second routine in this section first checks to see if the fire key was pressed (when we will have K=1). If it was, a bullet is fired, starting with a bleep noise, up the screen. The position of the dot is tested against the alien's position to see if it has hit him. If it hasn't it keeps going until it gets to the top of the screen. If it hits the alien the flag HIT is set to 1.

The third routine (listing 3.7c) checks whether the fire key was pressed and, if so, fires a laser using the LINE command. This is accompanied by a bleep. The horizontal position of the laser is checked against AX and if the two coincide then HIT is set to 1.

Section 8: EXPLOSIONS

This section will be called from the control program only if a HIT has occurred. In this section we wish to make some sort of noise and possibly give a visible signal to indicate that this has happened. You may choose any of the first ten routines (listings 3.8a – j), as your main routine for this section. It is very important to remember, whichever one of these you choose, that listing 3.8k must also be added if you have a FIREing game. This routine will wipe out the dead alien.

Listing 3.8a

```
1800 REM **********
1801 REM *EXPLOSION*
1802 REM * BURBLE  *
1805 REM **********
1810 F$="T5002L12AGBEDFADF#":F$=
F$+F$
1820 PLAY F$
1890 RETURN
```

Listing 3.8b

```
800 REM ************
1801 REM * EXPLOSION *
1802 REM *   TRILL   *
1805 REM ************
1810 F$="T150O5L12CF#CF#CF#CF#CF
#CF#CF#CF#"
1820 PLAY F$
1890 RETURN
```

Listing 3.8c

```
1800 REM ***********
1801 REM *EXPLOSION*
1802 REM *   HIT   *
1805 REM ***********
1810 F$="T150O1L30GF#FED#DC#C"
1820 PLAY F$
1890 RETURN
```

Listing 3.8d

```
1800 REM ************
1801 REM * EXPLOSION *
1802 REM *START AGAIN*
1805 REM ************
1810 F$="T30O3L6GEL18AL6GE"
1820 PLAY F$
1890 RETURN
```

Listing 3.8e

```
1800 REM ************
1801 REM * EXPLOSION *
1802 REM *ALIEN FIRED*
1805 REM ************
```

```
1810 F$="T100O4L8CC#DD#EFF#GF#FE
D#DC#C"
1820 PLAY F$
1890 RETURN
```

Listing 3.8f

```
1800 REM **************
1801 REM * EXPLOSION *
1802 REM * RASPBERRY *
1805 REM **************
1810 F$="T50O1L4CF#CF#CF#CF#CF#"
1820 PLAY F$
1890 RETURN
```

Listing 3.8g

```
1800 REM **************
1801 REM * EXPLOSION *
1802 REM *   FLASH   *
1805 REM **************
1810 FOR Q=1 TO 10
1820 SCREEN1,0:SOUND150,
1830 SCREEN1,1:SOUND100,
1840 NEXT Q
1890 RETURN
```

Listing 3.8h

```
1800 REM **************
1801 REM * EXPLOSION *
1802 REM *   BLIPS   *
1805 REM **************
1810 FOR Q=1 TO 5
1820 SOUND100+RND(100),1
1840 NEXT Q
1890 RETURN
```

Listing 3.8i

```
1800 REM **************
1801 REM * EXPLOSION *
1802 REM *    BLIP    *
1805 REM **************
1810 FOR Q=1 TO 5
1820 SOUND150,1
1840 NEXT Q
1890 RETURN
```

Listing 3.8j

```
1800 REM **************
1801 REM * EXPLOSION *
1802 REM *   SIREN    *
1805 REM **************
1810 FOR Q=1 TO 5
1815 FOR Y=100 TO 130 STEP 5
1820 SOUNDY,1
1825 NEXT Y
1830 FOR Y=130 TO 100 STEP -5
1835 SOUNDY,1
1840 NEXT Y
1850 NEXT Q
1870 RETURN
```

This routine *must* be added if there is FIREing in your game.

Listing 3.8k

```
1880 IF F=1 THEN PUT(XA,AY-8)-(X
A+7,AY-1),AB,PSET:DD=1:ALIENS=AL
IENS-1
```

All the routines in this section use either the SOUND or PLAY command to produce the noises. PLAY is the more versatile by far

of these two commands and is hence used most often, but SOUND can still be very effective for certain effects.

The routine 3.8k simply PUTs back the background over the alien to erase him when he has blown up. It then sets the flag DD to 1 to tell the move routine to start a new alien, and finally, takes one off the number of ALIENS left.

Section 9: SCORE ROUTINES

If you have HIT an alien there is obviously the question of how much was SCORED. Well, these routines are fairly simple and give you the choice of SCOREing one, five or ten points for each alien you HIT. If yours is a dodging game then you will not want to SCORE at all so use the first routine (listing 3.9a). Otherwise you can pick any of the other routines from this section.

Listing 3.9a

```
1900 REM ******************
1901 REM *SCORING ROUTINE*
1902 REM *    NOT USED     *
1905 REM ******************
1990 RETURN
```

Listing 3.9b

```
1900 REM ******************
1901 REM *SCORING ROUTINE*
1902 REM *     1 POINT     *
1905 REM ******************
1910 IF HIT=0 OR F=0 THEN 1990
1920 SCORE=SCORE+1
1990 RETURN
```

Listing 3.9c

```
1900 REM ******************
1901 REM *SCORING ROUTINE*
1902 REM *    5 POINTS    *
1905 REM ******************
1910 IF HIT=0 OR F=0 THEN 1990
1920 SCORE=SCORE+5
1990 RETURN
```

Listing 3.9d

```
1900 REM ********************
1902 REM *SCORING 10 POINTS*
1905 REM ********************
1910 IF HIT=0 OR F=0 THEN 1990
1920 SCORE=SCORE+10
1990 RETURN
```

All the routines in this section (except 3.9a) check to see if you are playing a FIREing game and whether you just scored a HIT on an alien. If you did then the required number of points are simply added to your SCORE.

Section 10: FUEL AND AMMUNITION

In some games you can fail your mission because of lack of FUEL or AMMO. If you do not wish to alter the quantities of either of these then you can select just the first routine from this section. If, however, you wish to alter the quantity use listing 3.10b and add the appropriate listings from the rest of this section. Note that you can only choose one out of each of the pairs of routines (listings 3.10d and 3.10e for AMMO and listings 3.10g and 3.10h for FUEL) for increasing the quantities, since these occupy the same line numbers.

Listing 3.10a

```
2000 REM **********************
2002 REM *FUEL AND AMMUNITION*
2003 REM *      NOT USED      *
2005 REM **********************
2090 RETURN
```

Listing 3.10b

```
2000 REM ************************
2001 REM *                      *
2002 REM * FUEL AND AMMUNITION *
2003 REM *                      *
2004 REM ************************
2090 RETURN
```

Listing 3.10c

```
2017 REM ****************
2018 REM *DECREASE AMMO*
2019 REM ****************
2020 IF K=1 THEN AMMO=AMMO-1
```

Listing 3.10d

```
2027 REM ****************
2028 REM *INCREASE AMMO*
2029 REM ****************
2030 IF HIT=1 THEN AMMO=AMMO+1
```

Listing 3.10e

```
2027 REM ****************
2028 REM * DOUBLE AMMO *
2029 REM ****************
2030 IF HIT=1 THEN AMMO=AMMO*2
```

Listing 3.10f

```
2037 REM ******************
2038 REM * DECREASE FUEL *
2039 REM ******************
2040 IF K=1 THEN FUEL=FUEL-1
```

Listing 3.10g

```
2047 REM ****************
2048 REM *INCREASE FUEL*
2049 REM ****************
2050 IF HIT=1 THEN FUEL=FUEL+1
```

Listing 3.10h

```
2047 REM ***************
2048 REM * RESET FUEL *
2049 REM ***************
2050 IF HIT =1 THEN FUEL=10
```

The routines for decreasing FUEL or AMMO check whether the fire key was pressed and take off one unit per shot. The routines for increasing FUEL or AMMO check whether you scored a HIT and if you did they can add one unit, double the number of units left, or reset the quantities to full.

Section 11: STATUS & DISPLAY

After all the excitement so far with explosions, lasers blasting away and your SCORE rapidly rising we need to make sure the relevant information is displayed on the screen. Of course if you have been peacefully dodging a plague of flies in the desert you won't need to display anything but you must still put a routine into this slot. For a dodging game use listing 3.11a. For a FIREing game start with listing 3.11b, which will display the SCORE. If

your game includes a varying supply of FUEL or AMMO then add either listing 3.11c or 3.11d.

Listing 3.11a

```
2100 REM ******************
2102 REM *STATUS DISPLAY *
2103 REM *    NOT USED    *
2107 REM ******************
2190 RETURN
```

Listing 3.11b

```
2100 REM ******************
2102 REM *STATUS DISPLAY *
2103 REM *                *
2107 REM ******************
2110 '
2117 REM ****************
2118 REM *DISPLAY SCORE*
2119 REM ****************
2120 P$=STR$(SCORE)+"  ":XS=144:Y
S=176:GOSUB 9900
2190 RETURN
```

Listing 3.11c

```
2127 REM ***************
2128 REM *DISPLAY AMMO*
2129 REM ***************
2130 P$=STR$(AMMO)+"  ":XS=40:YS=
176:GOSUB 9900
```

Listing 3.11d

```
2137 REM ***************
2138 REM *DISPLAY FUEL*
2139 REM ***************
2140 P$=STR$(FUEL)+"  ":XS=224:YS
=176:GOSUB 9900
```

These routines are quite straightforward. They convert the values in question into strings and pass them, along with their required screen positions, to the PUT STRING routine. The screen positions are carefully arranged to ensure that messages do not overlap even if they are all being used.

Section 12: CHECK FOR END OF GAME

In this section we provide all the checks that see whether we have reached an end of game situation. If any of these checks are found to be true then we must set the flag FIN to 1. This will signal to the control program that it is time to stop playing the game and move on to the end of game display routine. If FIN is zero the control program will go back to the MOVE/FIRE subroutine. From this section we need to type in the main routine (listing 3.12a) and at least one of the remaining five routines. After all, we want to stop the game sometime!

The first of the checks (listing 3.12b) is on whether the number of ALIENS is down to 0: This could be used in a dodging game, for instance, where you win if you managed to dodge ten aliens. For a dodging game you will also need to test for a collision with listing 3.12c.

You might decide that you lose if more than 3 aliens get PAST and over-run your base. This could be tested for by adding listing 3.12d.

The final two listings deal with testing for running out of AMMO or FUEL if you wish these to indicate the end of the game.

Listing 3.12a

```
2200 REM ************************
2202 REM *CHECK FOR END OF GAME*
2205 REM ************************
2290 RETURN
```

Listing 3.12b

```
2220 IF ALIENS=0 THEN FIN=1
```

Listing 3.12c

```
2230 IF HIT=1 THEN FIN =1
```

Listing 3.12d

```
2240 IF P=3 THEN FIN=1
```

Listing 3.12e

```
2250 IF FUEL=0 THEN FIN=1
```

Listing 3.12f

```
2210 IF AMMO=0 THEN FIN=1
```

These routines all test variables or flags set elsewhere in the program and when one of the conditions is satisfied, they set the FIN flag to 1.

Section 13: END OF GAME DISPLAY

If the FIN flag has been set for some reason then we want to stop the game. We could just stop the game. This is the simplest and most obvious method. You will then have to type RUN to play again. Listing 3.13a is just such a QUICK STOP routine. However it might be nice to end up with a polite question as to whether or not you wish to try again. Well, since it costs nothing to be polite, listing 3.13b is a POLITE STOP. If yours is a SCOREing game, then you might also like to add listing 3.13c to tell you what SCORE has been achieved.

Listing 3.13a

```
2300 REM **************
2302 REM * STOP GAME *
2303 REM *QUICK STOP *
2305 REM **************
2310 STOP
2390 RETURN
```

Listing 3.13b

```
2300 REM **************
2302 REM * STOP GAME *
2303 REM *POLITE STOP*
2305 REM **************
2310 CLS
2330 PRINT@235,"ANOTHER GO?":A$=
INKEY$
2340 A$=INKEY$:IF A$="" THEN 234
0
2350 IF A$<>"N" THEN 2390
2360 STOP
2390 RETURN
```

Listing 3.13c

```
2320 PRINT@136,"YOUR SCORE WAS";
SCORE
```

The first routine in this section is fairly obvious. The second, more polite, routine clears the TEXT SCREEN and PRINTs the question ANOTHER GO? (Note that the TEXT SCREEN comes up as soon as you PRINT anything on it – you do not have to ask for it specifically.) The computer now waits for you to press a key, and if you press anything except "N" it RETURNs to the control program, which will start again, otherwise it STOPs. If you add in listing 3.13c then your score will be PRINTed on the screen above the question.

CHAPTER 4

Starting to Write Your Own Games

Hopefully, you've had a lot of fun making your own games from the routines in Chapter 3. But there is no reason why you shouldn't make up your own routines to extend the variety available for a section. Perhaps you are wishing there was a different background available, or a different explosion. Well, in this chapter we will take a good look at how we could change or completely replace the routines given in Chapter 3. Let's go through the routines as listed and see how they really work.

INSTRUCTIONS

These are fairly easy to alter. In BASIC a line can either be a command on its own or several commands separated by colons. These commands, plus whatever follows them up to the next colon (or the end of the line), are known as statements. There are a lot of PRINT statements in this section so let's take a look at some of these first. Basically, anything inside the quote marks gets printed on the screen, so for the example game you could change line 1060 from:

```
1060 PRINT @ 258,"USE 0 TO FIRE": F=1
```

to:

```
1060 PRINT @ 258, "USE 0 TO FIRE YOUR LASER":F=1
```

As you can see the only differences are inside the quotes and this would simply PRINT the new message on the screen instead of the original message. Altering the position of the message on the screen is also fairly simple. The screen on your Dragon is divided up into 16 rows of 32 columns, making 512 PRINT positions. These are numbered 0 to 511, with the first row going from 0 to 31, the second from 32 to 63, and so on. The @ sign means "at", and we can PRINT "at" any of these 512 positions by typing PRINT

@ (position). Let's display a 'GOOD LUCK!' message on the screen. We need to pick a line number that will go in between the ones that are already used. 1075 seems like a good choice. We want to PRINT the message lower than the last instruction so we'll put it two rows below it and centred, so it will be at position 365. (If you want to think in terms of rows and columns then if you number them from 0 to 15 and from 0 to 31 you can work out the position as 32*row number + column number. So, our new line of BASIC is:

```
1075 PRINT @ 365,"GOOD LUCK!"
```

And it's as simple as that.

BACKGROUNDS

Most of the backgrounds in this book involve the RaNDom distribution of shapes on a plain background. We will see how the shapes are made up in Chapter 6 so let's not worry about that now. Line 1360 (in listings 3.2b to 3.5e) controls the number of objects, in this case 25, that we have on the screen. If you wanted to make the game a bit harder (for a dodging game), or just wanted more shapes on the screen (for a firing game), then you could change the 25 to a larger number. This would put extra waves or stars (or whatever other shape is used) on the screen.

Line 1370 PUTs the shapes on to the screen at an x position which is a RaNDom multiple of 8 in the range 0 to 248 (X2=RND(32)*8–8) and a y position which is a RaNDom multiple of 8 in the range 0 to 168 (Y2=RND(22)*8–8).

The routines that draw skylines are slightly different from the others. In the first case you cannot have a different number of buildings across the bottom of the screen as there is only space for 32 characters, each 8 points in width. Also we do not want the buildings PUT at RaNDom positions. In listing 3.5g what we do is to use RND() to determine height of the buildings at random. So in line 1360 I goes from 0 to 248 STEP 8 for the X position, while in line 1365 J goes from 8 TO 8*RND(2) STEP 8 for the Y position. Hang on, this means our buildings will be dangling down from

the top of the screen! We need to use 168-J as the Y position to ensure that the storeys of the buildings are built up from the bottom.

In listing 3.5h we use two objects as buildings and then PUT them in pairs across the bottom of the screen. In this case we have I going from 0 TO 248 STEP 16 (instead of STEP 8). We could have PUT four buildings at a time and then we would have needed to go up in steps of 32.

If you are not playing a dodging game you can fill the screen with anything you like – surreal space invaders?

PLAYER AND ALIEN GRAPHICS

These will be dealt with separately in Chapter 6.

SET UP AND RE-START

Although this is a small routine, and there is only one routine in this section, there are quite a few major changes you can make.

You could change the number of aliens you have to fight – either reducing the number to make the game easier to beat, or increasing the number to make the game more difficult. At present it is set to ten but if you wanted to have fifteen aliens then you could change line 1420 to:

```
1420 ALIENS=15
```

You could also change your starting level of ammunition to make the game harder or easier in much the same way as altering the number of aliens. This is done in line 1440. Suppose you wished to make the game a lot easier by having 100 bullets/bombs to start with then you could change the line to:

```
1440 AMMO=100
```

The amount of fuel can also be changed. Increasing the number will make the game easier (your fuel will last longer) and

decreasing the number will make the game harder (you will run out of fuel sooner). Line 1450 sets the amount of FUEL you have to start with.

Remember that you might also want to change the routines in the section where fuel and ammunition are altered.

MOVE AND FIRE

There is very little in this section that can be changed without knowing exactly what you're doing. However it is worth looking at how we move the characters around. Before a character is PUT anywhere the background is always GOT first, so that when we erase the player by PUTting the background back we automatically ensure that the background is restored. This way we avoid wiping out any stars, etc., when we pass in front of them.

DETECT A HIT

These routines use simple animation techniques to move a missile up the screen from the player. When you are more experienced at writing programs you could rewrite these to, say, move the missile in any of four directions from the player instead of just restricting firing to up the screen. For the moment we won't bother to look at these routines in detail.

EXPLOSIONS

You can really have a lot of fun in this section, creating different noises to go with your game. Once you have grasped the methods of producing music you could even add a signature tune to your game! Sound on the Dragon can be achieved with one of two commands – PLAY or SOUND. SOUND can vary the pitch over most of the audible range and can vary the duration from about $^1/_{16}$ of a second to as long as you wish This is fine for many applications, but for music and really spectacular effects we need to PLAY. The PLAY command is always followed by a string

containing numbers and letters which represent most of the musical parameters such as tempo, note length, volume, rests and, of course, pitch. This is not the place for a music lesson, but suffice it to say that you can use this command to make an enormous variety of sounds.

The routine that flashes the screen does so by quickly alternating between the two available colour sets with the SCREEN command.

SCORING

There are one or two alterations that can be made to this routine, mainly concerned with the way scoring is carried out. The amount your score is incremented can be changed quite easily, and this has already been done in the routines listed. If you wanted to increase your score by 20 points for every alien shot down then you could change line 1920 to:

```
1920 SCORE=SCORE+20
```

A more complex routine would give a higher score the earlier you managed to hit the alien, so you would need a routine that converts a low y coordinate to a high score. When the alien reaches you AY will be 168, so if you subtract AY from 168 you will score more for hitting the alien early on. You can divide the result by, say, 8 to prevent ridiculously high scores, and take its INT value to avoid decimal scores. The new line would look like this:

```
1920 SCORE=SCORE+INT((168-AY)/8)
```

You can experiment with different ratios as much as you like, of course.

FUEL AND AMMUNITION

The routines in this section can be altered to increase or decrease the amounts by which fuel and bullets are used up. The value for

ammunition is stored in the variable AMMO and the amount of fuel is stored in the variable FUEL. Another variable used in these routines is HIT, which, as we have seen already, is used to indicate whether or not you managed to hit an alien (it is set to 0 if you didn't, and 1 if you did). This variable is used in dodging games to indicate that you have crashed. A variable used in this way, as an indicator, is called a *flag*.

REDISPLAY THE SCREEN

There are not many changes you can make in this section, however you could make a couple of alterations if you wanted to make the game faster and a little harder. If you leave out lines 2130 and 2140 then you won't know what levels of fuel and ammunition you have left. That will make the game much harder to play, because you won't know if you're just about to run out of fuel or bullets. Because the Dragon now has two less jobs to do, the program runs faster. It's like someone taking away some of your typing – you work much faster.

A more complicated change would be to put in a high score variable, so that you could see whether you were improving or not. This is more complex because you have to put extra lines in other routines as well before you can print it on the screen.

First you must decide what to call the variable in which you keep the high score. If you look at Appendix One at the back of this book, you will see a list of the variables and their uses. Choose one that is *not* in the list. It is best to choose something meaningful such as HISCORE.

Don't forget to keep track of which extra line numbers you use or you could find yourself overwriting something you wanted to keep. It's best to look at the listing on the computer before adding any new lines.

In the END OF GAME routine we have to write a line which compares the high score with the score you have just received. If the latter is the greater it must alter the high score to be equal to your score. This means we will have to use an IF statement and we can write the statement out (in English) as IF (MY SCORE) IS

GREATER THAN (HIGH SCORE) THEN LET (HIGH SCORE) EQUAL (MY SCORE). This is very easy to convert to the BASIC language and is equivalent to:

```
IF SCORE>HISCORE THEN HISCORE=SCORE
```

Which goes to show how much like a shortened form of English the BASIC language is.

Having done this we must return to altering the routine to update the screen display. We need an extra line in here to PRINT the high score on the screen and we could put one in that looked like this:

```
2185 P$=STR$(HISCORE)+" ":XS=80:YS=176:GOSUB
9900
```

You can of course put the high score anywhere you like, but try to ensure it doesn't interfere too much with the rest of the game. Look again at the section on changing the instructions if you're not sure how to go about it.

CHECK FOR END OF GAME ROUTINES

These routines would be quite awkward to change but perhaps when you have some more experience you could devise a system which tells you why the game has ended. You would need to examine the different variables to see why the game ended. For example, a variable could hold the numbers 1 to 4 to show that:

1 – The aliens overran your base
2 – You shot down all the aliens
3 – You ran out of fuel
4 – You ran out of bullets

The variable that flags the end of the game is FIN so you could change the value that is assigned to it in each line of the CHECK routine. In the END OF GAME routine you could PRINT a different message, dependent on the value of FIN, that would tell the player why he had won or lost the game. These messages could be

slotted in between lines 2320 and 2330 and would look like this:

```
2321 IF FIN=1 THEN PRINT @ 197, "YOU WERE
OVERRUN BY ALIENS"
2322 IF FIN=2 THEN PRINT @ 197, "YOU SHOT
DOWN ALL THE ALIENS"
2323 IF FIN=3 THEN PRINT @ 197, "YOU RAN OUT OF
FUEL"
2324 IF FIN=4 THEN PRINT @ 197, "YOU RAN OUT OF
AMMUNITION"
```

Of course you could still add these lines even if you were using the impolite ending.

That concludes this chapter on altering the routines in the arcade games. Remember to take everything one step at a time, testing as you go so that you don't have to look through reams of alterations to find one mistake. It is also a good idea to keep CSAVEing the program after making alterations just in case the power goes off for any reason. Whether it's because grandma fell over the plug or because the blackout for World War Three has come, you're going to be very annoyed if several hours of unsaved typing suddenly changes into a blank screen.

CHAPTER 5

Further Explanations and Understanding BASIC

Some of the BASIC used in creating the routines has already been explained in the previous chapter but from now on we will be using and referring to more advanced techniques for programming. If you have been working through the book and want to learn more about the technicalities of BASIC then carry on with this chapter. In it we will examine the way BASIC actually works.

MORE PRINT ITEMS

In previous routines you may have noticed that various parts of the PRINT line are separated by semi-colons(;), these are, logically enough, called PRINT separators, but they are not there just to break up the statement, they have a definite purpose. Try the following short routine:

```
10 FOR X=1 TO 9
20 PRINT "!";
30 NEXT X
```

and notice the result – it should look like this:

```
!!!!!!!!!
```

Now try the routine without the semi-colon at the end of the PRINT statement:

```
10 FOR X=1 TO 9
20 PRINT "!"
30 NEXT X
```

The result is different – it looks like this:

Now try it with a comma:

```
10 FOR X=1 TO 9
20 PRINT "!",
30 NEXT X
```

Once more the result is different:

From the above, we have noticed several things which can be summarised as follows:

1. A semi-colon causes the PRINT items to be PRINTed one after another with no spaces in between.

2. A comma causes the PRINT items to be PRINTed one in each half of the screen.

3. The absence of either a semi-colon or a comma causes an automatic line feed which means that each PRINT item is PRINTed on a separate line.

If you try PRINTing numbers on the Dragon the rule for semi-colons may at first appear to go awry. Try the following:

```
10 FOR X=1 TO 9
20 PRINT X;
30 NEXT X
```

and you will get:

1 2 3 4 5 6 7 8 9

Why the spaces? This is because whenever the Dragon PRINTs numbers it always puts a space in front of them. This can be quite useful sometimes for tabulating results, but at other times it can be quite a nuisance! Either way, you should keep it in mind whenever you plan to PRINT numbers.

We have already used PRINT @ and seen how you can use this to PRINT anywhere on the screen, but there is another form – PRINT TAB() – which PRINTs at a specific column on the screen, rather like the TAB on a typewriter. You cannot use it to backspace on a line, however. If you typed:

```
10 PRINT TAB(20);"MY NAME IS";TAB(18);"FRED"
```

You would get

```
   My name is
Fred
```

The computer did the best it could in the circumstances! If you use a TAB number bigger than 31 (31 would give you the 32nd column on the screen because the first column is numbered 0) the computer will move to the next line and PRINT at the column which is 32 less than the TAB number you gave it, so that:

```
10 PRINT TAB(2); "WHAT'S UP"; TAB(34);"DOC"
```

would produce the following:

```
WHAT'S UP
DOC?
```

PRINTING VARIABLES

So far, we have only seen how to PRINT items enclosed in quotes ("") but we can also PRINT variables. We've already met numeric variables, used to store numbers such as the score and the screen

positions. They can be PRINTed in just the same way as things in quotes:

```
10 PRINT @0,SCORE
```

and if, for instance, the number 2200 is stored in SCORE, 2200 will be PRINTed in the top left-hand corner of the screen (preceded by that space we mentioned earlier). This is how we displayed the player's score at the end of the game:

```
2320 PRINT@136,"YOUR SCORE WAS";
SCORE
```

Notice that because the computer PRINTs a space in front of numbers we didn't need to put one after "WAS" inside the quotes.

There is another type of variable called a *string variable* and these contain characters instead of numbers. We will talk about these more in Chapter 7.

PRINTING ON THE GRAPHICS SCREEN

In the background routines are lines which place headings, such as "SCORE:", on the screen. They do not use PRINT because this command only works with the TEXT screen, which is completely separate from the GRAPHICS screen. Instead they give a string variable the characters for the heading and pass this (P$), along with the screen co-ordinates where we want the headings to go, to the PUT STRING subroutine at line 9900. This routine takes each character in turn from P$, GETs it from the collection of characters defined in the initialisation section of the program, and PUTs it on screen. We will see more on GET and PUT in Chapter 6.

PROGRAM CONTROL

Programs are controlled by asking questions and then directing flow to different parts of the program depending on the answer. You ask a question in BASIC by saying IF (CONDITION)THEN

(ACTION). CONDITION can be many things and you can use the following symbols (or *operands*) to make the test.

= Equals
> Greater than
< Less than
<= Less than or equal to
>= Greater than or equal to
<> Not equal to

Here are a few examples of their use:

```
20 IF A>B THEN GOTO 40
30 PRINT A
40 PRINT B
```

If A holds a bigger number than B, the program will jump to line 40 and only the value of B will be PRINTed, but if A holds a number equal to, or less than B then the values of both A and B will be PRINTed, because the program will carry on with line 30, instead of jumping to line 40.

```
20 IF A$<>B$ THEN STOP
30 PRINT A$
40 PRINT B$
```

In this case, we are testing to see if two string variables contain the same letters. If they don't, the program will stop, and if they are identical, the program will PRINT out the contents of A$ and B$.

```
20 IF A-B=5 THEN GOSUB 1000
30 GOSUB 2000
```

Line 20 checks to see if the result of subtracting the value of B from the value of A is equal to 5, and if it is the program will jump to the subroutine at line 1000, and will proceed to the subroutine at line 2000 if it isn't. The contents of A and B are not affected by this. ACTION can also be many things and we have already used three different actions in the previous examples. Here are some more:

```
20 IF Y<>0 THEN LET Y=0
```

This will reset Y to 0 if it is some other number.

```
30 IF M$="Y" THEN SOUND 75,1
```

This will produce a sound if the string variable M$ contains the letter Y.

```
40 IF (X+Y)/2<Z THEN CLS
```

Line 40 will clear the screen (CLS) if the result of adding the values in X and Y together and then dividing by two is less than the value of Z. Calculations in brackets are always performed first; without the brackets multiplication and division will be done first and the value of Y would first be divided by 2, and then added to the value of X. This can be seen from the following example:

```
(3+7)/2 = 10/2=5
```

whereas:

```
3+7/2 = 3 + 3.5 = 6.5
```

Two or more conditions can be tested at one time, using AND, OR and NOT.

AND

This is very similar to its meaning in English and, when it is used, the specified action will only happen when all the conditions are true. For example:

```
20 IF X=Y AND M$="YES" THEN STOP
```

or even:

```
30 IF H=67 AND X$<>"NO" AND A-B=Z+W AND
G$=K$ THEN CLS: GOTO 20
```

In the above line, the screen will only be cleared, and the program then jump to line 20, when all four conditions are met.

OR

This is really the opposite of AND, in that the action will be

performed if either condition is true, but not if they are both false. For example:

```
20 IF X=0 OR B=1 THEN SOUND 75,1
```

The note will sound if X holds the value of 0, or B holds the value 1, or if both are true.

NOT

NOT can sometimes be very useful. Just as in English, it is used to switch true (which the Dragon takes as –1), and false (0) values. We can say, for example:

```
1510 IF NOT (A=B) THEN GOTO 90
```

The condition A=B is true if A does equal B, false otherwise. Putting NOT before the condition reverses the values the Dragon uses as true and false, so that A is NOT equal to B then the program will go to line 90, since NOT (A=B) is true.

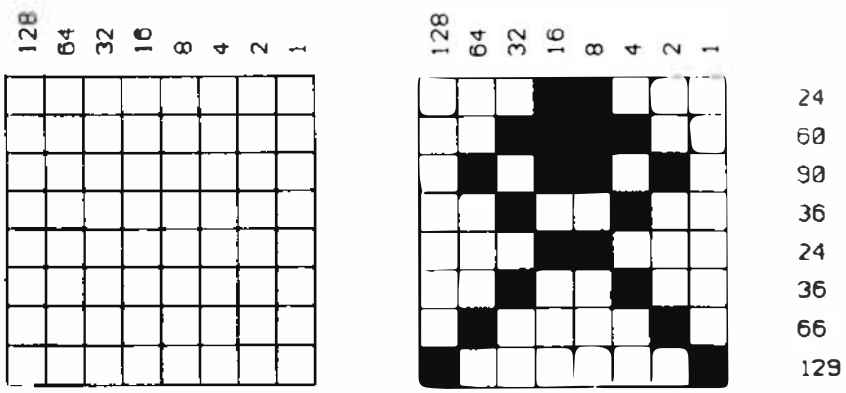
Right, that's enough BASIC for now. In the next chapter we will deal with all the fun and frolics of how to define characters like the aliens and the player.

CHAPTER 6

Character Graphics

In this chapter we will be taking a look at one of the most important parts of programming games: Graphics characters. We have already used many of these without bothering to worry about how they are made up or how they work. Let's take a look at one of the routines that sets up a graphics character. For example, look at the routines that define aliens. As you can see, they are all very similar, in fact the only differences are in the lines starting with REM and the line starting with DATA. The line starting with REM, if you remember, just means REMark or REMinder, so after REM you can write anything you like – the program will ignore this statement – it's just there to help us humans understand the program.

Next we'll deal with the DATA lines. You may have noticed that there are always eight numbers in each DATA statement This is because of the way in which our little characters are made up. They are drawn on an eight by eight grid of squares. We have to draw our character on a grid like this, and remember, you can only use whole squares, not parts – it's all or nothing.



Above is a blank grid and next to it we've drawn a new alien, so that you can see how to turn him into a set of numbers. If you look again, you can see the row of numbers across the top of the grid. Each square in the grid has a value, and that is given by the

number above it, so to get the number for each line, you start at one end, and if the square is blank you move to the next, and if the square is blacked in you add it to your total. The first line is hence:

16 + 8 = 24

and the rest of the lines are as follows:

32 + 16 + 8 + 4 = 60
64 + .16 + 8 + 2 = 90
32 + 4 = 36
16 + 8 = 24
32 + 4 = 36
64 + 2 = 66
128 + 1 = 129

Now you know enough to be able to design your own graphics characters so that you can have different aliens or players. You can even design new shapes for use in the background. What we really need to know, though, is how the routine actually works. So let's go back over it in greater detail.

1. The REM statement we have already dealt with, and we know that REM is short for REMinder or REMark.

2. There is a number system called *binary*, based on zeroes and ones instead of our *decimal* system which is based on the digits 0 to 9. This binary system is the only number system that your microcomputer directly understands. It *seems* to understand decimal numbers but that is only because there is a program inside that converts everything – even the words – into binary numbers.

We can use this system to represent our graphics character, by putting a one where a square is blacked in and a zero where a square is left blank. In this way we would get 8 binary digits. If you look in Appendix Four you will see that a series of eight of these binary digits are equivalent to a decimal number between 0 and 255, and the numbers of the DATA statements are the decimal equivalents of these binary numbers. In computing we call a binary digit a BIT for short. Eight bits make what is called one BYTE, and your computer's memory is divided up into bytes.

Your Dragon has 65536 bytes of memory, 32768 of which are available for you to put information into. Within a byte, as you move along from right to left each bit is worth double that of the one before it – hence the sequence of numbers at the the top of our graphics grid:

128, 64, 32, 16, 8, 4, 2, 1

If each bit was set to 1, the number held in that byte would be 255 decimal, or 11111111 in binary.

3. The next statement sets two variables, PG and N, and then GOSUBs to the routine at line 9000 which does the actual defining of the character.

4. This routine is very short, but it manages to do a lot of work. The first line just arranges for us to see graphics pages PG, PG+1, PG+2, and PG+3. You will notice that in our character definitions we always set PG=2 so you will see pages 2,3,4 and 5. This line is not really necessary, but we might like to see what's going on. The next line sets the first address in memory which we are going to write to. We want to define our characters on to page 5 of the graphics screen memory so that we can GET them afterwards. This page starts at memory location 7680, and this is what ST (the STart variable) is set to. The next statement is a FOR/NEXT statement. We have used these quite a lot but they have not yet been fully explained. The FOR statement always goes with a NEXT statement and it is a way of counting and performing an action a specified number of times, so, for example:

```
1220 FOR I=0 TO 7
.
.
.
1230 NEXT I
```

will carry out the instructions in between eight times – the count starts from zero and goes up to seven in steps of one. It can be made to go up or down in other steps, so that:

```
10 FOR K = 10 TO 20 STEP 2
.
.
.
50 NEXT K
```

will mean that K will hold the following values as the program goes round the loop:

10, 12, 14, 16, 18, 20

and the FOR . . . . NEXT loop:

```
10 FOR T = 100 TO 10 STEP - 10
.
.
.
50 NEXT T
```

will give the following as values of T:

100, 90, 80, 70, 60, 50 40, 30, 20, 10

You will notice that on the line after the FOR statement in line 9010, there is another FOR statement. This is perfectly respectable, and it is called "nesting loops" (sounds cosy doesn't it). It is OK as long as you remember that the loops must not overlap, so the last loop "opened" with a FOR must be the first loop "closed" with a NEXT. These two loops are both closed in line 9050, but with just one NEXT statement This is a feature special to the Dragon. It allows us to say NEXT Y,CH instead of NEXT Y:NEXT CH, although we could still do it this way if we wanted.

5. Now for what is going on inside the loops. Well, first we READ the number in the DATA statement into the variable CD (for CoDe), then we check to see if the iNVerse flag (NV) is set (NV is set or reset in the initialisation section of the program) and if it is we set CD to 255-CD. This has the effect of changing all the zeros to ones and vice versa – think about it. The next line POKEs this value (CD) directly into memory, at the address worked out by the expression ST+224*RN+CH+32*Y. Don't worry if you can't see why this expression works. It's not obvious and you don't need to understand it to use it. In fact, the terms 224*RN and CH are not

necessary for the character defining routines, they are only necessary when you are defining more than one character at a time, such as in the initialisation part of the program which defines the entire alphabet, and the numbers 1 to 9, into page 6, so that we can put scores, etc., on to the graphics screens during the game. (That's what all those DATA statements are for in lines 500 to 584!).

6. When the subroutine at 9000 has RETURNed control to the "ALIEN SKULL", or whatever, routine, we need to GET the character into an array. We will be discussing arrays in their more usual context in the next chapter. For now just think of it as putting aside an area of memory in which we can store our graphics character, ready to dump him on screen at a moment's notice. A word of warning here: you can only dimension an array once within a program – attempting to do it a second time will cause an error report and stop your program. For this reason it is often a good idea to include all DIM statements in the initialisation part of the program. As we only visit the character definition routines once, however, we can put them here and it does make it clearer what they are being used for.

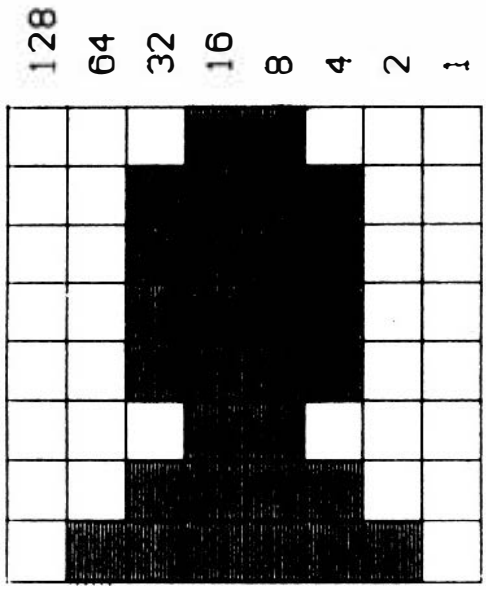
The GET instruction takes the form: GET(X,Y)-(X+P,Y+Q),A,G where X and Y are the screen co-ordinates of the top left corner of the rectangle of the screen which we want to GET. P is the width of the rectangle minus 1, Q is the height of the rectangle minus 1, and A is the name of the array into which we are GETting the bit of screen in question. GET's partner in crime is PUT and it looks very similar: PUT(X,Y)-(X+P,Y+Q),A,PSET. This PUTs the area of screen previously GOT into A, back on to the screen at position X,Y. The PSET means "put it back just as you got it". If we wished we could use PRESET, AND, OR, or NOT instead of PSET.

PRESET: This PUTs an inverted picture on to the screen. By inverted we do not mean upside down, but that the foreground becomes background and vice versa.
NOT: This ignores the array completely and just inverts whatever is on that part of the screen at the time.
AND: This compares each point in the array with each point already on the bit of screen in question and sets the point on

screen if they are both set, otherwise it resets it.
OR: This is like AND except that it will set the screen point if either of the points are set, otherwise it resets it.

Well, that's how the character definition routines work. Of course you could do all this without using using READ, DATA, FOR and NEXT, but the program would be badly styled, look very cumbersome and be difficult to change.

Where else can we use graphics characters? Well, if you remember we said we would be creating some more firing routines in this chapter, and it is in these routines that we can make further use of graphics characters. We will define a bomb and a fireball which we can GET/PUT up the screen, and then we will see how to utilise these in a firing routine. First, a picture of a bomb:



The numbers for it would be:

16 + 8 = 24
32 + 16 + 8 + 4 = 60
32 + 16 + 8 + 4 = 60
32 + 16 + 8 + 4 = 60
32 + 16 + 8 + 4 = 60
16 + 8 = 24
32 + 16 + 8 + 4 = 60
64 + 32 + 16 + 8 + 4 + 2 = 126

Now then, we need to decide where to put the lines which will define the character. We do not want to put them in the firing routine because that would mean redefining the character each time we fired a shot. This is very inefficient, and it would slow down the program intolerably. A good place would be in the alien graphics routine, and it would look like this:

```
1183 DATA 24,60,60,60,60,24,60,126
1185 GOSUB 9000
1187 DIMC(1):GET(0,144)-(7,151),C,G
```

Notice that this time we don't need to set PG or N because they will still be set from the alien definition. This would mean that array C now contained a bomb! Here is the listing which can be slotted into the firing routine part of the game if you wish:

Listing 6.1

```
1700 REM ****************
1701 REM *CHECK FOR HIT*
1702 REM *     BOMB      *
1705 REM ****************
1710 HIT=0: IF K=0 THEN 1790
1715 SOUND100,1
1720 BX=PX: BY=160
1725 GET(BX,BY)-(BX+7,BY+7),N,G:
PUT(BX,BY)-(BX+7,BY+7),C,PSET
1730 BY=BY-8
1735 PUT(BX,BY+8)-(BX+7,BY+15),N
,PSET
1740 GET(BX,BY)-(BX+7,BY+7),N,G:
PUT(BX,BY)-(BX+7,BY+7),C,PSET
1745 IF XA=PX AND AY=BY THEN HIT
=1:GOTO 1760
1750 IF BY>0 THEN 1730
1760 PUT(BX,BY)-(BX+7,BY+7),N,PS
ET
1790 RETURN
```

If you want a fireball instead of a bomb just change the DATA to this:

```
1183 DATA 0,0,36,24,126,24,36,0
```

By following the above format, you should be able to create many different types of missile.

Following this text is a utility program (a utility program is one that helps you to design and create other programs) to help you make up your own characters and work out the DATA for them without using reams of paper and wearing out lots of pencils. (After all, what's a computer for, if not to make life easier?) Your position in the grid is shown by a cross, which you can move with the arrow keys. When you reach a square you want to change – either from black to white or white to black – press C and it will change. As you create your character you will see a real-size version appear at the bottom of the screen, and under the word DATA you will see the numbers that you will need to put into the DATA statement in your definition routine.

This is a longish program, so be careful how you type it in. It would be a good idea to CSAVE it for future use.

Character Editor Listing

```
10 PCLEAR5:PMODE4,2:PCLS1
20 PG=2:N=44:NV=1:GOSUB 9000
30 DIMC(1):DIMO(1):DIMN(1):DIMDT
(7)
40 GET(88,152)-(95,159),C,G
50 GOSUB 1000
60 GOSUB 2000
100 A$=INKEY$:IF A$="" THEN 100
110 IF A$="C" THEN GOSUB 3000:GO
TO 100
120 IF A$=CHR$(8) THEN IF CX>96
THEN F=F*2:CX=CX-8:GOSUB 2000:GO
TO 100
130 IF A$=CHR$(9) THEN IF CX<152
 THEN F=F/2:CX=CX+8:GOSUB 2000:G
OTO 100
```

```
140 IF A$=CHR$(10) THEN IF Y<7 T
HEN AD=AD+32:Y=Y+1:CY=CY+8:GOSUB
 2000:GOTO 100
150 IF A$=CHR$(94) THEN IF Y>0 T
HEN AD=AD-32:Y=Y-1:CY=CY-8:GOSUB
 2000
160 GOTO 100
500 DATA 0,60,66,66,126,66,66,0
502 DATA 0,124,66,124,66,66,124,
0
504 DATA 0,60,66,64,64,66,60,0
506 DATA 0,120,68,66,66,68,120,0
508 DATA 0,126,64,124,64,64,126,
0
510 DATA 0,126,64,124,64,64,64,0
512 DATA 0,60,66,64,78,66,60,0
514 DATA 0,66,66,126,66,66,66,0
516 DATA 0,62,8,8,8,8,62,0
518 DATA 0,2,2,2,66,66,60,0
520 DATA 0,68,72,112,72,68,66,0
522 DATA 0,64,64,64,64,64,126,0
524 DATA 0,66,102,90,66,66,66,0
526 DATA 0,66,98,82,74,70,66,0
528 DATA 0,60,66,66,66,66,60,0
530 DATA 0,124,66,66,124,64,64,0
532 DATA 0,60,66,66,114,74,60,0
534 DATA 0,124,66,66,124,68,66,0
536 DATA 0,60,64,60,2,66,60,0
538 DATA 0,254,16,16,16,16,16,0
540 DATA 0,66,66,66,66,66,60,0
542 DATA 0,66,66,66,66,36,24,0
544 DATA 0,66,66,66,90,102,66,0
546 DATA 0,66,36,24,24,36,66,0
548 DATA 0,130,68,40,16,16,16,0
550 DATA 0,126,4,8,16,32,126,0
552 DATA 0,0,0,0,0,0,0,0
562 DATA 999,999,999,999,999
564 DATA 0,24,36,44,52,36,24,0
```

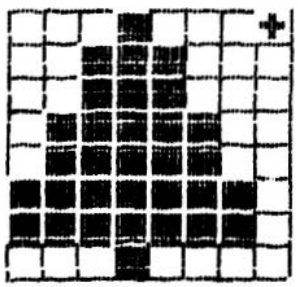
```
566 DATA Ø,8,24,8,8,8,28,Ø
568 DATA Ø,24,36,8,16,32,6Ø,Ø
57Ø DATA Ø,24,36,24,4,36,24,Ø
572 DATA Ø,8,24,4Ø,72,124,8,Ø
574 DATA Ø,6Ø,32,56,4,36,24,Ø
576 DATA Ø,28,32,56,36,36,24,Ø
578 DATA Ø,6Ø,4,8,16,32,32,Ø
58Ø DATA Ø,24,36,24,36,36,24,Ø
582 DATA Ø,24,36,36,28,4,56,Ø
584 DATA Ø,Ø,8,Ø,Ø,Ø,8,Ø
586 DATA Ø,24,24,126,126,24,24,Ø
999 REM **DISPLAY GRID**
1ØØØ PMODE4,1:PCLSNV:SCREEN1,Ø
1Ø1Ø COLORØ,1:FOR X=96 TO 16Ø ST
EP 8
1Ø2Ø LINE(X,64)-(X,127),PSET
1Ø3Ø NEXT X
1Ø4Ø FOR Y=64 TO 128 STEP 8
1Ø5Ø LINE(96,Y)-(159,Y),PSET
1Ø6Ø NEXT Y
1Ø7Ø P$="DATA":XS=56:YS=48:GOSUB
 99ØØ
1Ø9Ø FOR YS=64 TO 12Ø STEP 8
11ØØ XS=72:P$="Ø":GOSUB 99ØØ:NEX
T YS
111Ø P$="C TO CHANGE":XS=168:YS=
96:GOSUB 99ØØ
112Ø CX=96:CY=64:OX=CX:OY=CY:AD=
6160:F=128:Y=Ø
113Ø GET(OX,OY)-(OX+7,OY+7),O,G
114Ø RETURN
1999 REM **CROSS**
2ØØØ PUT(OX,OY)-(OX+7,OY+7),O,PS
ET
2Ø1Ø GET(CX,CY)-(CX+7,CY+7),O,G
2Ø2Ø PUT(CX,CY)-(CX+7,CY+7),C,PS
ET
```

```
2030 OX=CX:OY=CY
2040 RETURN
2999 REM **CHANGE**
3000 PUT(CX,CY)-(CX+7,CY+7),0,PS
ET
3010 PUT(CX,CY)-(CX+7,CY+7),C,NO
T
3020 GET(CX,CY)-(CX+7,CY+7),0,G
3030 IF PPOINT(CX+3,CY+3)=0 THEN
 DT(Y)=DT(Y)+F ELSE DT(Y)=DT(Y)-
F
3035 A=DT(Y):IF NV=1 THEN A=255-
A
3040 POKE AD,A
3050 P$=STR$(DT(Y)):P$=RIGHT$(P$
,LEN(P$)-1)
3060 XS=80-8*LEN(P$)
3070 YS=64+8*Y
3075 COLOR1,1:LINE(56,YS)-(80,YS
+7),PSET,BF
3080 GOSUB 9900
3090 PUT(CX,CY)-(CX+7,CY+7),C,PS
ET
3095 RETURN
9000 PMODE4,PG:SCREEN1,0:REMthis
 is just so you can see it happe
ning
9010 ST=7680+1536*(PG-2)
9020 FOR CH=0 TO N-1:RN=INT(CH/3
2)
9030 FOR Y=0 TO 7:READ CD:IF CD=
999 THEN Y=7:GOTO 9050
9035 IF NV=1 THEN CD=255-CD
9040 POKE ST+224*RN+CH+32*Y,CD
9050 NEXTY,CH
9055 RETURN
9899 REM *PRINT STRING*
```

```
9900 IF P$="" THEN RETURN
9910 A$=LEFT$(P$,1):P$=RIGHT$(P$
,LEN(P$)-1)
9920 IF A$=" " THEN YG=144:XG=20
8:GOTO 9950
9930 YG=144:AS=ASC(A$)-65:IF A$<
"A" THEN YG=152:AS=ASC(A$)-48
9940 XG=8*AS
9950 GOSUB 9960:XS=XS+8:GOTO 990
0
9960 PMODE4,2:GET(XG,YG)-(XG+7,Y
G+7),N,G
9970 PMODE4,1:PUT(XS,YS)-(XS+7,Y
S+7),N,PSET:RETURN
```

Here is a sample screen from the program:

```
DATA
 1 5
 5 6
 2 5 5 6
1 2 2 4 5 6
1 2 2 4 4 6
2 5 5 4 4 6
  5 5 4 4 6
  1 6 6
```



```
C TO CHANGE
```

CHAPTER 7

Arrays and Adventures

By now you should be quite familiar with the idea of a variable. We have been using variables to store scores, the number of aliens and many other things.

These variables are called simple variables – they contain one number only. If you want to store another number you either use another variable or overwrite what was stored in the first one.

For example, going back to our cheesecake example for a moment, suppose you wanted to keep track of your profit on cheesecake over a week. One way would be to have seven different variables, one for each day. Then your program would be seven times as long, as it would be doing the same calculation on seven different variables, and at the end it would print out your profits for the seven days with a routine like this:

```
100 PRINT P1
110 PRINT P2
120 PRINT P3
130 PRINT P4
140 PRINT P5
150 PRINT P6
160 PRINT P7
```

Well, it would work, but there is a better way – using *arrays*.

An array is in effect a collection of variables with the same name. But unlike simple variables, you cannot LET them equal anything until you have told the computer that you are going to use them. You do this with the DIM statement, and this is called DIMensioning the array.

In our example, we need an array with seven "elements". Now it is one of the peculiar facts in computing that computers usually start counting from 0, not from 1 like humans, so it counts the elements in an array as 0,1,2,3,4,5,6, and so on. So, to get seven elements we only need to go up to 6. (This may seem odd at first, but you will soon get used to it, and it is actually very useful in

some circumstances.) DIM P(6), then, will create an array of seven elements, and we refer to them as P(0), P(1), and so on. Now, the really clever bit about arrays is that the number inside the brackets (called the subscript) can be another variable! This means that we could store our cheesecake profits for each day in P(0) to P(6) and to PRINT them out we could use a routine like this:

```
100 FOR N=0 TO 6
110 PRINT P(N)
120 NEXT N
```

and we think you will agree that this is much better than the previous method.

This array is said to have one DIMension, with seven elements – a one by seven array. We have used this DIMension for the days of the week. However, you can have arrays with as many DIMensions as you like. Let's say we wanted to find out which flavours made the most popular cheesecake (so we can maximise our profits!) We would set up a two DIMensional array. One DIMension would have seven elements as before and the other DIMension would have as many elements as we have flavours. Let's keep it simple and say that we have strawberry, chocolate, blackcurrant and plain – four flavours in all. Our array would be seven by four, so our DIM statement would be DIM P(6,3). The first day's profit for strawberry cheesecake would go into P(0,0), for chocolate it would be P(0,1), for blackcurrant P(0,2) and for plain P(0,3). The second day's profit would be in P(1,0), P(1,1), P(1,2) and so on. To PRINT these out at the end of the program, you would be able to use a routine like this:-

```
100 FOR DAY=0 TO 6
110 FOR FLAVOUR= 0 TO 3
120 PRINT P(DAY,FLAVOUR);" ";
130 NEXT FLAVOUR
140 PRINT
150 NEXT DAY
```

This would give us a neat table – flavours across and days down, with 28 figures in all. Can you imagine using 28 variables and 28 PRINT statements to do the same job?

STRING VARIABLES AND STRING ARRAYS

Variables and arrays which contain numbers are called "numeric" variables or arrays. This is not just a pointless piece of jargon; it distinguishes them from another, quite different, type of variable – a *string variable* – and another type of array – a *string array.*

We have used string variables in our arcade game to get scores, etc., on to the Dragon's high resolution graphics screen, but we have not really looked at them very closely

Whereas a numeric variable holds a number, a string variable holds a string, and the same goes for arrays. This is the major difference between the two types and it is a very important difference. A string is a collection of characters. The computer recognises a collection of characters as a string when they are between quotes. So, "FRED" is recognised by the computer as a string, but FRED without the quotes is assumed to be a numeric variable.

The important thing to remember is that you cannot mix the two types. If, for example, you tried to say LET A="FRED" the computer would give you the error report ?TM ERROR which stands for Type Mismatch, which is just what it was. String variables can have the same kind of names as numeric variables, but they always have a $ sign on the end to show that they are string variables. So, LET A$="FRED" would put the string "FRED" into the variable A$, and PRINT A$ would result in FRED appearing on the screen. LET A$=10, however, would give ?TM ERROR again, because you cannot put a number into a string variable.

Of course, digits are only characters, and they can be put in strings, so LET A$="10" is fine. But remember it is a string not a number and so, if you then tried to say LET X=2+A$ you would once again get ?TM ERROR. There are, however, two functions which allow you to change a number into a string and vice versa. LET A$=STR$(10) is quite acceptable, and would result in A$ containing the string " 10". Notice the space. This is why the computer PRINTs a space before numbers – because it uses the STR$ function itself to convert the numbers into strings before

printing them. LET X=VAL(A$) is also perfectly OK, and if A$ contained the string "10" (or " 10") then X would be assigned the value 10.

We can use string functions together with the two DIMensions of P$(6,3) to format the output from our cheesecake profits routine. We can use the STR$ function to first turn the numbers into strings, and then use another function which gives us the length of a string (LEN) to get all the numbers lined up. The program would look like this:-

```
100 FOR DAY=0 TO 6
110 FOR FLAVOUR =0 TO 3
120 LET A$=STR$ (P(DAY,FLAVOUR))
130 PRINT TAB(FLAVOUR*8 - LEN(A$)+8);A$
140 NEXT FLAVOUR
150 PRINT
160 NEXT DAY
```

Arrays and strings are used a lot in adventure games and this is what we'll take a look at next. In the adventure game given in the next chapter we will see multi-DIMensional arrays used to keep track of things like the player's position and the contents of rooms. The program in the next chapter builds up into a game in much the same way that the sections of Chapter 3 built up into an arcade game. Although there are some differences the process is sufficiently similar not to need a full example program.

CHAPTER 8

Adventure Games, a Selection of Lego Bricks

In this chapter we are going to write an adventure game. We will begin as we did for our arcade game – by considering the basic building blocks of such a game. These are as follows:

1. Initialisation.
2. Assign the Inventories.
3. Give Instructions.
4. Create the Maze.
5. Describe Situation.
6. Player's INPUT.
7. Check INPUT is Legal.
8. Perform Instruction.
9. Computer Response.
10. Check for End of Game.
11. End of Game Message.
12. Round Again.

To begin with, we will write the simplest (well almost!) version of the game. We have chosen a scenario of a maze of dungeons. The player must search for a Crown of Emeralds left there aeons ago by the king of a long forgotton race of Troglodytes. The only "feature" of the game is that some of the passages from cave to cave have doors that are locked, and to open them the player needs to find and take a key, of which there are several lying about in the maze. To prevent things from becoming too easy we will not allow any keys to be used more than once – they will always either break or get stuck in the lock.

Step 1. Initialisation. OK! Let's build the game. Again we will begin with the initialisation and control program, which follows the sequence set out above. The pattern is a familiar one – for each block, we just GOSUB to a subroutine which does whatever

the blockname says it must do (eg. create a maze), just as in our arcade game.

The control program is shown in listing 8.1. Type it in exactly as shown.

Listing 8.1

```
10 REM ****************
20 REM *TREASURE TRAIL*
30 REM ****************
40 REM ** INITIALISE DRAGON **
50 CLEAR 500
60 DIMIA$(7,1):DIMIB$(10):DIMIC$
(2)
70 DIMRM$(35,5):DIMAJ$(4,1):DIMP
S(5)
80 FOOD=0:BATTERIES=0:MN=0
90 DEF FNR(DR)=(6 AND DR=3)-(6 A
ND DR=2)+(1 AND DR=4)-(1 AND DR=
5):DEF FND(DR)=DR-(DR=2 OR DR=4)
+(DR=3 OR DR=5)
100 REM ********************
110 REM *ASSIGN INVENTORIES*
120 REM ********************
130 GOSUB 1200
140 REM ***************
150 REM *INSTRUCTIONS*
160 REM ***************
170 GOSUB 1000
180 REM **REPEAT UNTIL user quit
s**
190 REM **************
200 REM *CREATE MAZE*
210 REM **************
220 GOSUB 1400:NR=1:PRINT@448,""
;
230 REM **REPEAT UNTIL game over
**
```

```
240 REM ***********
250 REM *NEW ROOM?*
260 REM ***********
270 ON NR+1 GOSUB 2020,2000
280 REM **************
290 REM *PLAYER INPUT*
300 REM **************
310 GOSUB 2500
330 REM ******************
340 REM *LEGAL MANOUVRE?*
350 REM ******************
360 GOSUB 3000:IF Q=1 THEN 650
420 REM ******************
430 REM *VALID MANOUVRE?*
440 REM ******************
450 IF LG=1 THEN GOSUB 4000
460 REM ******************
470 REM *DRAGON RESPONSE*
480 REM ******************
490 GOSUB 5000
500 REM **UNTIL game over**
510 IF LOST=0 AND WON =0 THEN 27
0
520 REM *************
530 REM *END OF GAME*
540 REM *************
550 IF LOST=1 THEN GOSUB 6000
560 IF WON=1 THEN GOSUB 6100
570 REM ***************
580 REM *ANOTHER GAME?*
590 REM ***************
600 INPUT"DO YOU WANT ANOTHER GA
ME";U$
610 IF LEFT$(U$,1)<>"N" THEN CLS
:GOSUB 6200:GOTO 220
620 REM *****
630 REM *END*
640 REM *****
650 END
```

This control program will remain the same no matter which version of the game we decide to write so we suggest that you CSAVE it for future use.

The lines numbered less than 100 simply initialise the computer. Line 50 CLEARs an area of 500 bytes at the top of memory for the exclusive use of strings. Lines 60 and 70 DIMension the arrays which the program will use, and line 90 defines two functions which will be used later in the program. These operations will be explained more fully in a later section.

Step 2. Assign Inventories. Now all we have to do is write the subroutines which are called from the main program. The first of these is given here.

Listing 8.2

```
1199 REM *ASSIGN INVENTORIES*
1200 FOR N=0 TO 2
1210 READ IA$(N,0),IA$(N,1)
1220 NEXTN
1230 FOR N=0 TO 6
1240 READ IB$(N)
1250 NEXT N
1300 DATA GO,0123,TAKE,56,OPEN,4
1310 DATA NORTH,SOUTH,EAST,WEST,
DOOR,KEY,EMERALDS
1330 DATA DAMP,MISERABLE,COLD,DA
RK,SCARY,OPPRESSIVE,SMALL,GLOOMY
,LARGE,DRAUGHTY
1340 FOR N=0 TO 4
1350 READ AJ$(N,0),AJ$(N,1)
1360 NEXT N
1370 RETURN
```

Type this in with the control program. This routine assigns the inventories. The inventories are arrays containing all the words (verbs and nouns) which the program needs to be able to understand.

Step 3. Give Instructions. Instructions are needed to tell the player what to do in the game and what the aim is. However it is essential not to give too much information here – it is supposed to be a game of exploration and adventure after all! Combine the lines of listing 8.3 with your program to date.

Listing 8.3

```
999 REM *INSTRUCTIONS*
1000 CLS:PRINT@3,"T R E A S U R
E  T R A I L"
1010 PRINT@64,"YOU ARE IN THE DU
NGEONS OF GORM,";
1020 PRINT"AND YOU ARE SEARCHING
 FOR A"
1030 PRINT"CROWN OF EMERALDS"
1040 PRINT TAB(3);"THE COMPUTER
UNDERSTANDS THE"
1050 PRINT"FOLLOWING VERBS:-";
1060 FOR N=0 TO 2:PRINT TAB(18);
IA$(N,0):NEXT N
1190 PRINT@484,"PRESS ANY KEY TO
 START";:A$=INKEY$
1195 A$=INKEY$:IF A$="" THEN 119
5
1196 CLS:RETURN
```

Step 4. Create the Maze. It is essential to make absolutely certain that you could win. This routine creates the maze and places the target, the Crown of Emeralds, and the player at RaNDom positions. There will always be a possible path from one to the other but the rest of the maze is decided at random. Listing 8.4 gives the simplest version of the maze creation routine.

Listing 8.4

```
1399 REM *SET STATUS, PLACE CROW
N & PLAYER*
1400 PRINT TAB(13);"WAIT"
```

```
1410 CE=RND(35)
1420 X=RND(35):IF X=CE THEN 1420
 ELSE PR=X
1429 REM *BUILD A PATH*
1430 R=RND(4):ON R GOSUB 1550,16
50,1700,1700
1440 L$="O":IF KEY>0 THEN L$=L$+
"D"
1460 L=LEN(L$):R=RND(L):O$=MID$(
L$,R,1)
1470 IF O$="D" THEN KEY=KEY-1
1480 Y=CE-X:IF Y>5 THEN RM$(X,3)
=O$:X=X+6:RM$(X,2)=O$:GOTO 1540
1490 IF Y<-5 THEN RM$(X,2)=O$:X=
X-6:RM$(X,3)=O$:GOTO 1540
1500 IF Y>0 AND (X+1)/6<>INT((X+
1)/6) THEN RM$(X,4)=O$:X=X+1:RM$
(X,5)=O$:GOTO 1540
1510 IF Y>0 THEN RM$(X,3)=O$:X=X
+6:RM$(X,2)=O$:GOTO 1540
1520 IF Y<0 AND X/6<>INT(X/6) TH
EN RM$(X,5)=O$:X=X-1:RM$(X,4)=O$
:GOTO 1540
1530 IF Y<0 THEN RM$(X,2)=O$:X=X
-6:RM$(X,3)=O$
1540 IF X=CE THEN GOSUB 1800:RM$
(CE,0)="E":RETURN ELSE GOTO 1430
1549 REM *DEPOSIT OBJECT*
1550 L$="N":IF RND(0)>.5 THEN L$
=L$+"K"
1600 L=LEN(L$):R=RND(L):O$=MID$(
L$,R,1)
1610 IF O$="K" THEN KEY=KEY+1
1640 RM$(X,0)=O$:RETURN
1650 RETURN
1700 RETURN
1799 REM *COMPLETE MAZE*
1800 OB$="NKN":L$="ODW"
```

```
1805 L=LEN(L$):O=LEN(O$)
1810 FOR N=0 TO 35:FOR M=2 TO 5
1820 IF RM$(N,M)="" THEN RM$(N,M
)=MID$(L$,RND(L),1)
1825 GOSUB 1950
1830 NEXT M
1840 IF N<6 THEN RM$(N,2)="W"
1850 IF N>29 THEN RM$(N,3)="W"
1860 IF (N+1)/6=INT((N+1)/6) THE
N RM$(N,4)="W"
1870 IF N/6=INT(N/6) THEN RM$(N,
5)="W"
1890 IF RM$(N,0)="" THEN RM$(N,0
)=MID$(O$,RND(O),1)
1930 NEXT N
1940 RETURN
1949 REM **COMPLEMENTARY DOORS**
1950 IF M=2 THEN IF N>5 THEN RM$
(N-6,3)=RM$(N,M):RETURN
1960 IF M=3 THEN IF N<30 THEN RM
$(N+6,2)=RM$(N,M):RETURN
1970 IF M=4 THEN IF (N+1)/6<>INT
((N+1)/6) THEN RM$(N+1,5)=RM$(N,
M):RETURN
1980 IF M=5 THEN IF N/6<>INT(N/6
) THEN RM$(N-1,4)=RM$(N,M):RETUR
N
1990 RETURN
```

Step 5. Describe Situation. Describe the situation and surroundings to the player. At this point we embark on our adventure, and it is to this point that the program will return after our latest attempt at derring-do has met with total failure. When we enter each cave the computer describes it and the objects therein are detailed. It is this section which must produce the excessive verbiage that is a pre-requisite of adventure games. This routine produces a lot of pleonastic purple prose with much repetition and even tautology. Much of the information is, of

course, apocryphal. This task is dealt with by listing 8.5 which is given below. Once again we must key in the whole lot to add this function to our program.

Listing 8.5

```
1999 REM *DESCRIBE ROOM*
2000 N=RND(5)-1:M=RND(5)-1:A$=AJ
$(N,0):B$=AJ$(M,1)
2010 PRINT"YOU ENTER A ";A$;", "
;B$;" ":IF RND(0)>.5 THEN PRINT"
CAVE" ELSE PRINT"PASSAGE"
2020 PRINT"YOU SEE: ";:IF RM$(PR
,0)="N" THEN PRINT"NOTHING":GOTO
 2110
2040 IF RM$(PR,0)="E" THEN PRINT
 TAB(9);"THE CROWN OF EMERALDS"
2080 IF RM$(PR,0)="K" THEN ON RN
D(4) GOSUB 2160,2170,2180,2190
2110 FOR N=2 TO 5
2120 IF RM$(PR,N)="O" OR RM$(PR,
N)="D" THEN PRINT"THERE IS A DOO
R TO THE ";IB$(N-2)
2140 NEXT N
2150 NR=0:RETURN
2160 PRINT TAB(9);"A RUSTY KEY":
RETURN
2170 PRINT TAB(9);"A LARGE KEY":
RETURN
2180 PRINT TAB(9);"A GOLDEN KEY"
:RETURN
2190 PRINT TAB(9);"A WOODEN KEY"
:NR=0:RETURN
```

Step 6. Player's INPUT. We must allow the player to make his life and death decisions, carefully weigh the pros and cons and finally stumble blindly on. In other words, what do you want to do now? This routine will accept the commands of the player as typed in at

the keyboard. We must add this section to our rapidly growing program (bear in mind that computer adventures are always fairly large programs so you might want to save your program a few times along the way as an insurance policy against power failure). Listing 8.6 below contains the instructions to accept INPUT.

Listing 8.6

```
2499 REM **PLAYER INPUT**
2500 PRINT@511," ":PRINT@448,"";
2510 INPUT U$
2520 RETURN
```

Step 7. Check INPUT is Legal. This section checks that what you typed actually made sense, e.g. that it uses a legal verb like GO or TAKE, and in a legal manner, such as GO NORTH. (GO BANANAS is not legal input). Type this section in from the next listing.

Listing 8.7

```
2999 REM **LEGAL MANOUVRE?**
3000 LG=0: IF INSTR(1,U$,"QUIT")>
0 THEN Q=1:RETURN
3010 IF LEFT$(U$,1)=" " THEN U$=
RIGHT$(U$,LEN(U$)-1):GOTO 3010
3020 IF RIGHT$(U$,1)=" " THEN U$
=LEFT$(U$,LEN(U$)-1):GOTO 3020
3099 REM *INVENTORY SEARCH*
3100 Z$="":Z1$="":Z2$="":L=INSTR
(1,U$," ")
3105 IF L=0 THEN W1$=U$:GOTO3120
 ELSE W1$=LEFT$((U$),L-1):REM FI
ND VERB
3110 Z$=RIGHT$(U$,LEN(U$)-L):REM
 REST OF STRING
3120 X=-1:FOR V=0 TO 2
3130 IF W1$=IA$(V,0) THEN X=V:V=
2
3140 NEXT V
```

```
3150 IF X=-1 THEN RF=2:RETURN
3159 REM FIND OBJECT OF VERB
3160 L$=IA$(X,1):Y=-1
3170 FOR N=1 TO LEN(L$)
3180 M=VAL("&H"+MID$(L$,N,1))
3190 IF INSTR(1,Z$,IB$(M))>0 THE
N Z1$=IB$(M):Y=M:N=LEN(L$)
3200 NEXT N
3210 IF Y=-1 THEN RF=3 ELSE LG=1
3215 RETURN
```

Step 8. Perform Instruction. Well, we'll try to! Just because GO NORTH seems like a reasonable idea doesn't mean we can necessarily do it. There might be a blank wall in the way. There are so many possibilities that the routine is made up of many smaller routines which deal with individual words. This sort of further subdivision of tasks is a common feature in structured programs and makes both reading and writing them a lot easier.

Listing 8.8

```
3999 REM **VALIDITY CHECK**
4000 RF=0:OK=0:ON X+1 GOSUB 4100
,4200,4300
4010 IF OK=1 THEN RF=16
4030 RETURN
4099 REM **GO**
4100 REM
4105 DR=-1:FOR N=0 TO 3
4110 IF INSTR(1,Z$,IB$(N))>0 THE
N DR=N+2:N=3
4120 NEXT N
4125 IF DR=-1 THEN RF=14:RETURN
4130 IF RM$(PR,DR)="0" THEN OK=1
:PR=PR+FNR(DR):NR=1:RETURN
4140 IF RM$(PR,DR)="D" THEN RF=5
 ELSE RF=4
4150 RETURN
```

```
4199 REM **TAKE**
42ØØ REM
421Ø IF RM$(PR,Ø)<>LEFT$(IB$(Y),
1) THEN RF=6:RETURN
422Ø OK=1:PS(Y-5)=PS(Y-5)+1
423Ø RM$(PR,Ø)="N"
424Ø RETURN
4299 REM **OPEN**
43ØØ REM
43Ø5 IF PS(Ø)<1 THEN RF=7:RETURN
431Ø LD=Ø:DR=Ø:FOR N=Ø TO 3
432Ø IF DR=Ø THEN IF INSTR(1,Z$,
IB$(N))>Ø THEN DR=N+2
433Ø IF LD=Ø THEN IF RM$(PR,N+2)
="D" THEN LD=1
434Ø NEXT N
435Ø IF DR=Ø THEN RF=14:RETURN
436Ø IF LD=Ø THEN RF=8:RETURN
437Ø IF RM$(PR,DR)<>"D" THEN RF=
8:RETURN
438Ø RM$(PR,DR)="O":R=PR+FNR(DR)
:DR=FND(DR):RM$(R,DR)="O"
439Ø PS(Ø)=PS(Ø)-1:K=-1:OK=1:RET
URN
```

Step 9. Computer Response. By now we have figured out what has just happened with regard to the player's commands. We need to tell the player what he has (or more likely what he has not) achieved. Which response is given depends upon the RF response flag set by the previous routines. The choice of response is dealt with by listing 8.9 so add this to your program.

Listing 8.9

```
4999 REM **RESPONSE**
5ØØØ ON RF GOSUB 52ØØ,521Ø,522Ø,
523Ø,524Ø,525Ø,526Ø,527Ø,528Ø,52
9Ø,53ØØ,531Ø,532Ø,533Ø,534Ø,535Ø
,54ØØ,55ØØ,551Ø,552Ø
```

```
5010 IF PS(1)=1 THEN WON=1:RETUR
N
5020 IF LOST=1 THEN RETURN
5130 RETURN
5200 PRINT"WHAT?!":RETURN
5210 PRINT"I DO NOT KNOW THE VER
B ";W1$:RETURN
5220 PRINT"YOU CANNOT ";W1$;" ";
Z1$;Z$:RETURN
5230 PRINT"YOU CANNOT GO ";Z$:RE
TURN
5240 PRINT"THE DOOR IS LOCKED.":
RETURN
5250 PRINT"I SEE NO ";Z$;" HERE"
:RETURN
5260 PRINT"BUT YOU DO NOT HAVE A
 KEY!":RETURN
5270 PRINT"THERE IS NO LOCKED DO
OR TO OPEN HERE":RETURN
5330 PRINT"WHICH DIRECTION?":RET
URN
5350 PRINT"YOU ";U$
5360 IF K=-1 THEN PRINT"BUT";
5370 IF K=-1 THEN IF RND(0)>.5 T
HEN PRINT" THE KEY BREAKS":K=0 E
LSE PRINT" THE KEY GETS STUCK, Y
OU MUST    LEAVE IT BEHIND.":K=0
5390 RETURN
```

Step 10. Check for End of Game. If the game is neither won nor lost we have to loop back to the description routine (Step 5). This task has been incorporated into the control program, so there is no need to have a separate LISTing for this routine.

Step 11. End of Game Message. There are two routines in this section of which only one will be called, depending on whether the game has been won or lost. We will need to enter both, though, to guard against the possibility of success. So type in

both listings below (listings 8.11a and 8.11b).

Listing 8.11a

```
5999 REM **LOST**
6000 PRINT"YOU HAVE LOST THE GAM
E, BAD LUCK"
6010 RETURN
```

Listing 8.11b

```
6099 REM **WON**
6100 PRINT"CONGRATULATIONS - YOU
 HAVE","RECOVERED THE FABULOUS C
ROWN OF EMERALDS!!  WELL DONE!"
6110 FOR N=1 TO 200
6120 SOUND N,1
6130 NEXT
6140 RETURN
```

Step 12. Round Again. This section of the control program asks the player if he wants another game, and if he does then the subroutine in Listing 8.12 is called, which resets the flags and the possessions array (more about that later) and RETURNs to the control program, which promptly loops back to Step 3. If, on the other hand, the player answers "NO" then the control program carries on to its last statement – END. This produces the usual chirpy OK and puts the computer back in command mode.

Listing 8.12

```
6199 REM *RESET*
6200 WON=0:LOST=0
6210 FOR N=0 TO 5
6220 PS(N)=0
6230 NEXT N
6290 RETURN
```

And that's it! Your final listing should look like this:

Listing 8.13

```
1Ø REM ****************
2Ø REM *TREASURE TRAIL*
3Ø REM ****************
4Ø REM ** INITIALISE DRAGON **
5Ø CLEAR 5ØØ
6Ø DIMIA$(7,1):DIMIB$(1Ø):DIMIC$
(2)
7Ø DIMRM$(35,5):DIMAJ$(4,1):DIMP
S(5)
8Ø FOOD=Ø:BATTERIES=Ø:MN=Ø
9Ø DEF FNR(DR)=(6 AND DR=3)-(6 A
ND DR=2)+(1 AND DR=4)-(1 AND DR=
5):DEF FND(DR)=DR-(DR=2 OR DR=4)
+(DR=3 OR DR=5)
1ØØ REM **********************
11Ø REM *ASSIGN INVENTORIES*
12Ø REM **********************
13Ø GOSUB 12ØØ
14Ø REM ***************
15Ø REM *INSTRUCTIONS*
16Ø REM ***************
17Ø GOSUB 1ØØØ
18Ø REM **REPEAT UNTIL user quit
s**
19Ø REM **************
2ØØ REM *CREATE MAZE*
21Ø REM **************
22Ø GOSUB 14ØØ:NR=1:PRINT@448,""
;
23Ø REM **REPEAT UNTIL game over
**
24Ø REM ************
25Ø REM *NEW ROOM?*
26Ø REM ************
```

```
270 ON NR+1 GOSUB 2020,2000
280 REM ***************
290 REM *PLAYER INPUT*
300 REM ***************
310 GOSUB 2500
330 REM ******************
340 REM *LEGAL MANOUVRE?*
350 REM ******************
360 GOSUB 3000:IF Q=1 THEN 650
420 REM ******************
430 REM *VALID MANOUVRE?*
440 REM ******************
450 IF LG=1 THEN GOSUB 4000
460 REM ******************
470 REM *DRAGON RESPONSE*
480 REM ******************
490 GOSUB 5000
500 REM **UNTIL game over**
510 IF LOST=0 AND WON =0 THEN 27
0
520 REM **************
530 REM *END OF GAME*
540 REM **************
550 IF LOST=1 THEN GOSUB 6000
560 IF WON=1 THEN GOSUB 6100
570 REM ****************
580 REM *ANOTHER GAME?*
590 REM ****************
600 INPUT"DO YOU WANT ANOTHER GA
ME";U$
610 IF LEFT$(U$,1)<>"N" THEN CLS
:GOSUB 6200:GOTO 220
620 REM *****
630 REM *END*
640 REM *****
650 END
999 REM *INSTRUCTIONS*
```

```
1000 CLS:PRINT@3,"T R E A S U R
E  T R A I L"
1010 PRINT@64,"YOU ARE IN THE DU
NGEONS OF GORM,";
1020 PRINT"AND YOU ARE SEARCHING
 FOR A"
1030 PRINT"CROWN OF EMERALDS"
1040 PRINT TAB(3);"THE COMPUTER
UNDERSTANDS THE"
1050 PRINT"FOLLWING VERBS:-";
1060 FOR N=0 TO 2:PRINT TAB(18);
IA$(N,0):NEXT N
1190 PRINT@484,"PRESS ANY KEY TO
 START";:A$=INKEY$
1195 A$=INKEY$:IF A$="" THEN 119
5
1196 CLS:RETURN
1199 REM *ASSIGN INVENTORIES*
1200 FOR N=0 TO 2
1210 READ IA$(N,0),IA$(N,1)
1220 NEXTN
1230 FOR N=0 TO 6
1240 READ IB$(N)
1250 NEXT N
1300 DATA GO,0123,TAKE,56,OPEN,4
1310 DATA NORTH,SOUTH,EAST,WEST,
DOOR,KEY,EMERALDS
1330 DATA DAMP,MISERABLE,COLD,DA
RK,SCARY,OPPRESSIVE,SMALL,GLOOMY
,LARGE,DRAUGHTY
1340 FOR N=0 TO 4
1350 READ AJ$(N,0),AJ$(N,1)
1360 NEXT N
1370 RETURN
1399 REM *SET STATUS, PLACE CROW
N & PLAYER*
1400 PRINT TAB(13);"WAIT"
```

```
1410 CE=RND(35)
1420 X=RND(35):IF X=CE THEN 1420
 ELSE PR=X
1429 REM *BUILD A PATH*
1430 R=RND(4):ON R GOSUB 1550,16
50,1700,1700
1440 L$="O":IF KEY>0 THEN L$=L$+
"D"
1460 L=LEN(L$):R=RND(L):O$=MID$(
L$,R,1)
1470 IF O$="D" THEN KEY=KEY-1
1480 Y=CE-X:IF Y>5 THEN RM$(X,3)
=O$:X=X+6:RM$(X,2)=O$:GOTO 1540
1490 IF Y<-5 THEN RM$(X,2)=O$:X=
X-6:RM$(X,3)=O$:GOTO 1540
1500 IF Y>0 AND (X+1)/6<>INT((X+
1)/6) THEN RM$(X,4)=O$:X=X+1:RM$
(X,5)=O$:GOTO 1540
1510 IF Y>0 THEN RM$(X,3)=O$:X=X
+6:RM$(X,2)=O$:GOTO 1540
1520 IF Y<0 AND X/6<>INT(X/6) TH
EN RM$(X,5)=O$:X=X-1:RM$(X,4)=O$
:GOTO 1540
1530 IF Y<0 THEN RM$(X,2)=O$:X=X
-6:RM$(X,3)=O$
1540 IF X=CE THEN GOSUB 1800:RM$
(CE,0)="E":RETURN ELSE GOTO 1430
1549 REM *DEPOSIT OBJECT*
1550 L$="N":IF RND(0)>.5 THEN L$
=L$+"K"
1600 L=LEN(L$):R=RND(L):O$=MID$(
L$,R,1)
1610 IF O$="K" THEN KEY=KEY+1
1640 RM$(X,0)=O$:RETURN
1650 RETURN
1700 RETURN
1799 REM *COMPLETE MAZE*
```

```
1800 OB$="NKN":L$="ODW"
1805 L=LEN(L$):O=LEN(O$)
1810 FOR N=0 TO 35:FOR M=2 TO 5
1820 IF RM$(N,M)="" THEN RM$(N,M
)=MID$(L$,RND(L),1)
1825 GOSUB 1950
1830 NEXT M
1840 IF N<6 THEN RM$(N,2)="W"
1850 IF N>29 THEN RM$(N,3)="W"
1860 IF (N+1)/6=INT((N+1)/6) THE
N RM$(N,4)="W"
1870 IF N/6=INT(N/6) THEN RM$(N,
5)="W"
1890 IF RM$(N,0)="" THEN RM$(N,0
)=MID$(OB$,RND(O),1)
1930 NEXT N
1940 RETURN
1949 REM **COMPLEMENTARY DOORS**
1950 IF M=2 THEN IF N>5 THEN RM$
(N-6,3)=RM$(N,M):RETURN
1960 IF M=3 THEN IF N<30 THEN RM
$(N+6,2)=RM$(N,M):RETURN
1970 IF M=4 THEN IF (N+1)/6<>INT
((N+1)/6) THEN RM$(N+1,5)=RM$(N,
M):RETURN
1980 IF M=5 THEN IF N/6<>INT(N/6
) THEN RM$(N-1,4)=RM$(N,M):RETUR
N
1990 RETURN
1999 REM *DESCRIBE ROOM*
2000 N=RND(5)-1:M=RND(5)-1:A$=AJ
$(N,0):B$=AJ$(M,1)
2010 PRINT"YOU ENTER A ";A$;", "
;B$;" ":IF RND(0)>.5 THEN PRINT"
CAVE" ELSE PRINT"PASSAGE"
2020 PRINT"YOU SEE: ";:IF RM$(PR
,0)="N" THEN PRINT"NOTHING":GOTO
 2110
```

```
2040 IF RM$(PR,0)="E" THEN PRINT
 TAB(9);"THE CROWN OF EMERALDS"
2080 IF RM$(PR,0)="K" THEN ON RN
D(4) GOSUB 2160,2170,2180,2190
2110 FOR N=2 TO 5
2120 IF RM$(PR,N)="O" OR RM$(PR,
N)="D" THEN PRINT"THERE IS A DOO
R TO THE ";IB$(N-2)
2140 NEXT N
2150 NR=0:RETURN
2160 PRINT TAB(9);"A RUSTY KEY":
RETURN
2170 PRINT TAB(9);"A LARGE KEY":
RETURN
2180 PRINT TAB(9);"A GOLDEN KEY"
:RETURN
2190 PRINT TAB(9);"A WOODEN KEY"
:NR=0:RETURN
2499 REM **PLAYER INPUT**
2500 PRINT@511," ":PRINT@448,"";
2510 INPUT U$
2520 RETURN
2999 REM **LEGAL MANOUVRE?**
3000 LG=0:IF INSTR(1,U$,"QUIT")>
0 THEN Q=1:RETURN
3010 IF LEFT$(U$,1)=" " THEN U$=
RIGHT$(U$,LEN(U$)-1):GOTO 3010
3020 IF RIGHT$(U$,1)=" " THEN U$
=LEFT$(U$,LEN(U$)-1):GOTO 3020
3099 REM *INVENTORY SEARCH*
3100 Z$="":Z1$="":Z2$="":L=INSTR
(1,U$," ")
3105 IF L=0 THEN W1$=U$:GOTO3120
 ELSE W1$=LEFT$((U$),L-1):REM FI
ND VERB
3110 Z$=RIGHT$(U$,LEN(U$)-L):REM
 REST OF STRING
3120 X=-1:FOR V=0 TO 2
```

```
3130 IF W1$=IA$(V,0) THEN X=V:V=
2
3140 NEXT V
3150 IF X=-1 THEN RF=2:RETURN
3159 REM FIND OBJECT OF VERB
3160 L$=IA$(X,1):Y=-1
3170 FOR N=1 TO LEN(L$)
3180 M=VAL("&H"+MID$(L$,N,1))
3190 IF INSTR(1,Z$,IB$(M))>0 THE
N Z1$=IB$(M):Y=M:N=LEN(L$)
3200 NEXT N
3210 IF Y=-1 THEN RF=3 ELSE LG=1
3215 RETURN
3999 REM **VALIDITY CHECK**
4000 RF=0:OK=0:ON X+1 GOSUB 4100
,4200,4300
4010 IF OK=1 THEN RF=16
4030 RETURN
4099 REM **GO**
4100 REM
4105 DR=-1:FOR N=0 TO 3
4110 IF INSTR(1,Z$,IB$(N))>0 THE
N DR=N+2:N=3
4120 NEXT N
4125 IF DR=-1 THEN RF=14:RETURN
4130 IF RM$(PR,DR)="0" THEN OK=1
:PR=PR+FNR(DR):NR=1:RETURN
4140 IF RM$(PR,DR)="D" THEN RF=5
 ELSE RF=4
4150 RETURN
4199 REM **TAKE**
4200 REM
4210 IF RM$(PR,0)<>LEFT$(IB$(Y),
1) THEN RF=6:RETURN
4220 OK=1:PS(Y-5)=PS(Y-5)+1
4230 RM$(PR,0)="N"
4240 RETURN
4299 REM **OPEN**
```

```
4300 REM
4305 IF PS(0)<1 THEN RF=7:RETURN
4310 LD=0:DR=0:FOR N=0 TO 3
4320 IF DR=0 THEN IF INSTR(1,Z$,
IB$(N))>0 THEN DR=N+2
4330 IF LD=0 THEN IF RM$(PR,N+2)
="D" THEN LD=1
4340 NEXT N
4350 IF DR=0 THEN RF=14:RETURN
4360 IF LD=0 THEN RF=8:RETURN
4370 IF RM$(PR,DR)<>"D" THEN RF=
8:RETURN
4380 RM$(PR,DR)="O":R=PR+FNR(DR)
:DR=FND(DR):RM$(R,DR)="O"
4390 PS(0)=PS(0)-1:K=-1:OK=1:RET
URN
4999 REM **RESPONSE**
5000 ON RF GOSUB 5200,5210,5220,
5230,5240,5250,5260,5270,5280,52
90,5300,5310,5320,5330,5340,5350
,5400,5500,5510,5520
5010 IF PS(1)=1 THEN WON=1:RETUR
N
5020 IF LOST=1 THEN RETURN
5130 RETURN
5200 PRINT"WHAT?!":RETURN
5210 PRINT"I DO NOT KNOW THE VER
B ";W1$:RETURN
5220 PRINT"YOU CANNOT ";W1$;" ";
Z1$;Z$:RETURN
5230 PRINT"YOU CANNOT GO ";Z$:RE
TURN
5240 PRINT"THE DOOR IS LOCKED.":
RETURN
5250 PRINT"I SEE NO ";Z$;" HERE"
:RETURN
5260 PRINT"BUT YOU DO NOT HAVE A
 KEY!":RETURN
```

```
5270 PRINT"THERE IS NO LOCKED DO
OR TO OPEN HERE":RETURN
5330 PRINT"WHICH DIRECTION?":RET
URN
5350 PRINT"YOU ";U$
5360 IF K=-1 THEN PRINT"BUT";
5370 IF K=-1 THEN IF RND(0)>.5 T
HEN PRINT" THE KEY BREAKS":K=0 E
LSE PRINT" THE KEY GETS STUCK, Y
OU MUST    LEAVE IT BEHIND.":K=0
5390 RETURN
5999 REM **LOST**
6000 PRINT"YOU HAVE LOST THE GAM
E, BAD LUCK"
6010 RETURN
6099 REM **WON**
6100 PRINT"CONGRATULATIONS - YOU
 HAVE","RECOVERED THE FABULOUS C
ROWN OF EMERALDS!!  WELL DONE!"
6110 FOR N=1 TO 200
6120 SOUND N,1
6130 NEXT
6140 RETURN
6199 REM *RESET*
6200 WON=0:LOST=0
6210 FOR N=0 TO 4
6220 PS(N)=0
6230 NEXT N
6290 RETURN
```

I suggest that you CSAVE it immediately, before Grandma comes over again! When you've done that, RUN the program and see if you can find the Crown of Emeralds. GOOD LUCK!

ADDING FEATURES TO THE GAME

Now that we have our basic (no pun intended) game, we can think about adding various additional features to it. The features which can be added are limited only by your imagination (and the Dragon's memory of course). In the following sections we will see how to add five sets of additional features and by the end of this section you should have a reasonable idea of how the system works. Chapter 9 goes into the details of planning behind the various routines and by the end of that chapter you should be able to make your own alterations.

The program is designed so that features can be stuck on to it like Lego bricks – in a similar way to that in which we could add the ability to move up and down (for example) in our arcade game. There is a difference, however, due to the very nature of the game. Whereas in the arcade game, the routine for (say) moving the alien didn't give a hoot whether you could move up and down or not, in the adventure game alterations need to be made to almost all sections when a new feature is added. This is like the suggested alteration to allow a high score to be kept in the arcade game. In that case we had to go through the program altering quite a few of the routines to allow for the new feature. In just the same way we are going to have to go methodically through our adventure routines adding lines to each procedure.

FOOD AND STRENGTH

The first feature which we add we will call FOOD & STRENGTH. The first thing to do to it is figure out how this feature is related to the game scenario.

> We conjure up an evil force which pervades the chill caverns in which we are adventuring. The constant forces of cold and evil sap your physical and spiritual strength. To combat this we will have food which restores your fitness, body and soul. If however you do not find any food you will weaken and die!

The first section we have to add is given in listing 8.2a and this is adding to the inventory of objects in the dungeon. After all, if we

are going to put food in the maze, the computer is going to need to know about it. So, with your original game in memory, add in the following lines.

Listing 8.2a

```
12ØØ FOR N=Ø TO 3
123Ø FOR N=Ø TO 7
13ØØ DATA GO,Ø123,TAKE,567,OPEN,
4,EAT,76
131Ø DATA NORTH,SOUTH,EAST,WEST,
DOOR,KEY,EMERALDS,FOOD
```

The next thing we need to do is to tell the player about the presence of the food and the evil spirit. We add these lines to the instruction routine (listing 8.3a).

Listing 8.3a

```
1Ø6Ø FOR N=Ø TO 3:PRINT TAB(18);
IA$(N,Ø):NEXT N
1Ø7Ø PRINT@45Ø,"PRESS ANY KEY TO
 CONTINUE";:A$=INKEY$
1Ø8Ø A$=INKEY$:IF A$="" THEN 1Ø8
Ø
1Ø85 CLS
111Ø PRINT:PRINT"THERE IS AN EVI
L FORCE IN THE"
112Ø PRINT"DUNGEONS, AND IT DRAI
NS YOUR"
113Ø PRINT"STRENGTH, BUT IF YOU
ARE LUCKY"
114Ø PRINT"ENOUGH TO FIND THE MA
GIC FOOD,"
115Ø PRINT"YOU MAY SURVIVE."
```

We now need to set the strength of the player and also change the "create maze" routine so that it distributes some food around the

place as well as the other things. Add the couple of lines given in listing 8.4a to take care of this.

Listing 8.4a

```
1405 ST=1000
1580 IF RND(0)>.7 THEN L$=L$+"F"
```

The next thing to add is a line to make the computer tell us if there is actually some food in the room when we enter it. This is done by the line in listing 8.5a

Listing 8.5a

```
2060 IF RM$(PR,0)="F" THEN PRINT
 TAB(9);"SOME FOOD"
```

Of course at some stage we are going to want to EAT FOOD, so the computer must recognise this as a legal command. Add these lines to the legality checking procedures (listing 8.7a).

Listing 8.7a

```
3120 X=-1:FOR V=0 TO 3
3130 IF W1$=IA$(V,0) THEN X=V:V=
3
```

Similarly, we must add a routine to deal with an EAT command. Listing 8.8a has the lines to alter this section to call a new routine which deals with EATing.

Listing 8.8a

```
4000 RF=0:OK=0:ON X+1 GOSUB 4100
,4200,4300,4400
4399 REM **EAT**
4400 REM
4405 IF PS(Y-5)<1 THEN RF=10:RET
URN
4410 IF IB$(Y)<>"FOOD" THEN RF=1
1:RETURN
```

```
442Ø OK=1:PS(2)=PS(2)-1:ST=1ØØØ
443Ø RETURN
```

Finally to make life more fun add listing 8.9a to the response section. This will deal with PRINTing messages for the various eating habits possible and also add lines to warn the player of impending doom.

Listing 8.9a

```
5Ø3Ø ST=ST-3Ø:IF ST>5ØØ THEN 51Ø
Ø
5Ø4Ø IF ST>4ØØ THEN PRINT"YOUR S
TRENGTH IS FADING.":GOTO 51ØØ
5Ø5Ø IF ST>3ØØ THEN PRINT"YOU AR
E GETTING VERY WEAK":GOTO 51ØØ
5Ø6Ø IF ST>2ØØ THEN PRINT"YOUR S
TRENGTH IS EBBING FAST -  YOU CA
N'T GO ON MUCH LONGER":GOTO 51ØØ
5Ø7Ø IF ST>1ØØ THEN PRINT"IF YOU
 DON'T EAT SOON, YOU'VE  HAD IT
!":GOTO 51ØØ
5Ø8Ø IF ST>Ø THEN PRINT"YOU'RE O
N YOUR LAST LEGS MATE!":GOTO 51Ø
Ø
5Ø9Ø PRINT"YOU HAVE DIED OF EXHA
USTION!":LOST =1:RETURN
51ØØ RETURN
529Ø PRINT"YOU DO NOT HAVE THE "
;Z$:RETURN
53ØØ PRINT"YOU EAT THE ";Z$;". Y
OU","CHOKE TO DEATH!":LOST=1:RET
URN
```

That completes our first modification. RUN the program and convince yourself that no disasters have occurred.

TORCH AND BATTERIES

Our next modification deals with TORCH & BATTERIES supplies.

> You have a torch, and it's just as well, because the resident goblins don't like the light. What they would like, though, is to eat you. In fact they think that you're the best thing since sliced bread and preferably between slices as a snack for them. If your torch goes out then . . . The problem is that your batteries only last for 15 minutes, so unless you can find some more within that time, you will be eaten.

Well, that should put the pressure on a bit. Once again we will need to make modifications to several routines and as before the first place to add things is in the inventory. Add listing 8.2b.

Listing 8.2b

```
1230 FOR N=0 TO 8
1300 DATA GO,0123,TAKE,5678,OPEN
,4,EAT,768
1310 DATA NORTH,SOUTH,EAST,WEST,
DOOR,KEY,EMERALDS,FOOD,BATTERIES
```

We need to tell the player too (unless you are feeling heartless) so add the lines of listing 8.3b to the instructions routine.

Listing 8.3b

```
1160 PRINT:PRINTTAB(3);"ALSO, ON
LY YOUR TORCHLIGHT"
1170 PRINT"STOPS THE GOBLINS FRO
M EATING"
1180 PRINT"YOU, AND BATTERIES ON
LY LAST 15 ";"MINUTES!  GOOD LUC
K!"
```

Now we need to let the maze creation routine know there are extra objects to be distributed and we must also set the TIME variable to allow us to keep track of the number of turns you've had. Add listing 8.4b.

Listing 8.4b

```
1405 ST=1000:TIMER=0
1800 OB$="NKBN":L$="ODW"
```

Add the following to the room description routine.

Listing 8.5b

```
2070 IF RM$(PR,0)="B" THEN PRINT
 TAB(9);"SOME BATTERIES"
```

Listing 8.8b will save your b(e)acon if you pick up some batteries.

Listing 8.8b

```
4225 IF Y=8 THEN TIMER=0
```

Now give the computer a few words to say to tell you what a mess you're making of the game.

Listing 8.9b

```
5100 TM=15-INT(TIMER/3000)
5110 IF TM<1 THEN PRINT"YOUR TOR
CH HAS GONE OUT.","THE GOBLINS H
AVE EATEN YOU.":LOST=1:RETURN
5120 IF TM<6 THEN PRINT"YOUR BAT
TERIES HAVE ONLY";TM,"MINUTES PO
WER LEFT"
```

And that's it! Try the game again and see if you survive.

TROLLS, RUN, FIGHT AND TELEPORT

If you have not met the goblins yet you might be feeling lonely wandering around the maze, so how about introducing some trolls to keep you company!

> Trolls are lurking in the caves, and you never know when you are about to stumble upon one of these thin, rubbery and loathsome creatures. When you do you will have to decide whether to run or fight – if you try to do anything else he will attack you anyway. Fortunately you can usually beat him in a fight (although this might not be the case if your strength is low). If you find yourself cornered you have one other option – teleport – but heaven knows where you will end up if you use it, and it weakens you considerably to do so.

This modification has brought in the three new verbs RUN, FIGHT and TELEPORT and one new noun TROLL. Also it seems that tangling with a troll will affect a player's strength (as will teleporting) and that the presence of a troll will prevent him from picking up objects, opening doors etc. We begin to see how closely interrelated things become in adventure games as soon as they have more than one or two features. Implementing this feature follows the same pattern as the previous modifications.

First of all we add the new words in listing 8.2c.

Listing 8.2c

```
1200 FOR N=0 TO 6
1260 FOR N=0 TO 0
1270 READ IC$(N)
1280 NEXT N
1300 DATA GO,0123,TAKE,5678,OPEN
,4,EAT,768,RUN,,FIGHT,,TELEPORT,
1320 DATA TROLL
```

Next we give the player a cryptic hint of what is to come with the addition of listing 8.3c to the instructions.

Listing 8.3c

```
1060 FOR N=0 TO 6:PRINT TAB(18);
IA$(N,0):NEXT N
1090 PRINT"THERE ARE HIDDEN HAZA
RDS IN THE ";
1100 PRINT"DUNGEONS, WHICH YOU W
ILL HAVE TO";"OVERCOME."
```

Again we alter the maze routine to deposit a few trolls on its way to creating a maze.

Listing 8.4c

```
1649 REM *DEPOSIT HAZARD*
1650 L$="N":IF RND(0)>.5 THEN L$
=L$+"T"
1670 L=LEN(L$):R=RND(L):O$=MID$(
L$,R,1)
1690 RM$(X,1)=O$
1800 OB$="NKBN":L$="ODW":H$="NTN
"
1805 L=LEN(L$):O=LEN(O$):H=LEN(H
$)
1895 IF RM$(N,1)="" THEN RM$(N,1
)=MID$(H$,RND(H),1)
```

Next we make certain that the computer is actually going to tell you when you bump into a troll. Add listing 8.5c

Listing 8.5c

```
2020 PRINT"YOU SEE: ";:IF RM$(PR
,0)="N" AND RM$(PR,1)="N" THEN P
RINT"NOTHING":GOTO 2110
2090 IF RM$(PR,1)="T" THEN PRINT
 TAB(9);"A SMELLY TROLL":MN=1
```

As before we have to alter the routine to recognise valid commands so that it will accept these words. This is done by listing 8.7c.

Listing 8.7c

```
3120 X=-1:FOR V=0 TO 6
3130 IF W1$=IA$(V,0) THEN X=V:V=
6
3155 IF X>3 THEN LG=1:RETURN
```

You have probably noticed by now that the routine to perform verbs has one smaller section for each verb. The following listing contains routines for each of RUN, FIGHT and TELEPORT. Type in all of the listing given below.

Listing 8.8c

```
4000 RF=0:OK=0:ON X+1 GOSUB 4100
,4200,4300,4400,4700,4800,4900
4020 IF MN=1 AND RF<>12 THEN RF=
17
4100 IF MN=1 THEN RETURN
4200 IF MN=1 THEN RETURN
4300 IF MN=1 THEN RETURN
4400 IF MN=1 THEN RETURN

4699 **RUN**
4700 IF RM$(PR,1)="N" THEN RF=1:
RETURN
4705 FOR N=2 TO 5
4710 DR=0:IF RM$(PR,N)="0" THEN
DR=N:N=5
4720 NEXT N
4730 IF DR=0 THEN RF=20:RETURN
4740 PR=PR+FNR(DR):MN=0:RF=18:NR
=1
4750 RETURN
```

```
4799 REM **FIGHT**
4800 IF RM$(PR,1)="N" THEN RF=1:
RETURN
4810 MN=0:ST=ST-RND(100)
4820 RM$(PR,1)="N":RF=19
4830 RETURN

4899 REM **TELEPORT**
4900 NR=1:PR=RND(36)-1:ST=ST-200
4910 RF=16:MN=0
4920 RETURN
```

Finally we add listing 8.9c to the response section and we're finished.

Listing 8.9c

```
5400 PRINT"YOU TRY TO ";U$,"BUT
YOU ARE ATTACKED BY THE "
5420 IF RM$(PR,1)="T" THEN PRINT
"SMELLY TROLL"
5430 R=RND(100)+100:ST=ST-R
5440 RM$(PR,1)="N":MN=0
5450 IF ST<1 THEN LOST=1
5460 IF ST<1 THEN PRINT"YOU ARE
TOO WEAK TO KILL HIM, HEEATS YOU
"
5470 IF ST>0 THEN PRINT"YOU KILL
 HIM, HE CRUMBLES TO    DUST"
5480 RETURN
5500 PRINT"YOU RUN ";IB$(DR-2):R
ETURN
5510 PRINT"YOU FIGHT THE BEAST,
YOU STAVE  IN HIS SKULL.  HE CRU
MBLES TO   DUST":RETURN
5520 PRINT"YOU CANNOT RUN - THE
ENTRANCES  ARE CLOSED":RETURN
```

RUN the program again and look out for the trolls!

MONSTERS AND MAGIC DUST

Well, things are starting to hot up a bit now, aren't they? Now for another action packed feature.

> Monsters are found in the dungeons. These have the same effect on you as trolls but nastier. There are also little piles of magic dust in the labyrinths. If you throw this at the monsters then they will vanish.

This introduces another verb, THROW, and two more nouns, MONSTER and MAGIC DUST, so the first thing to do is add these to the inventory section.

Listing 8.2d

```
1200 FOR N=0 TO 7
1230 FOR N=0 TO 9
1260 FOR N=0 TO 1
1300 DATA GO,0123,TAKE,56789,OPE
N,4,EAT,7689,RUN,,FIGHT,,TELEPOR
T,,THROW,56789
1310 DATA NORTH,SOUTH,EAST,WEST,
DOOR,KEY,EMERALDS,FOOD,BATTERIES
,MAGIC DUST
1320 DATA TROLL,MONSTER
```

Add listing 8.3d to tell the player he can THROW things.

Listing 8.3d

```
1060 FOR N=0 TO 7:PRINT TAB(18);
IA$(N,0):NEXT N
```

As usual we have to change the maze creation routine so that it puts the new objects into the maze. We will also make sure that there is always a monster guarding the Crown of Emeralds. To do this we add the next listing.

Listing 8.4d

```
1540 IF X=CE THEN GOSUB 1800:RM$
(CE,0)="E":RM$(CE,1)="M":RETURN
ELSE GOTO 1430
1570 IF RND(0)>.6 THEN L$=L$+"M"
1630 IF O$="M" THEN MD=MD+1
1660 IF MD>0 THEN IF RND(0)>.8 T
HEN L$=L$+"M"
1680 IF O$="M" THEN MD=MD-1
1800 OB$="NKBN":L$="ODW":H$="NTM
N"
```

Now we need to add listing 8.5d to make sure the player is fully informed when he enters a room.

Listing 8.5d

```
2050 IF RM$(PR,0)="M" THEN PRINT
 TAB(9);"SOME MAGIC DUST"
2100 IF RM$(PR,1)="M" THEN PRINT
 TAB(9);"A HIDEOUS MONSTER":MN=1
```

Now we must extend the legal input check procedure to check that you have thrown something which is throwable (e.g. not a door). Add listing 8.7d to the program.

Listing 8.7d

```
3120 X=-1:FOR V=0 TO 7
3130 IF W1$=IA$(V,0) THEN X=V:V=
7
3155 IF X>3 AND X<7 THEN LG=1:RE
TURN
3210 IF Y=-1 THEN RF=3:RETURN
3215 REM
3220 IF X<7 THEN LG=1:RETURN
3229 REM FIND TARGET
3230 L=INSTR(1,Z$,Z1$):Z=-1
```

```
3240 Z$=LEFT$(Z$,L-1)+RIGHT$(Z$,
LEN(Z$)-(L+LEN(Z1$))):REM REMOVE
 OBJECT FROM Z$
3250 FOR N=0 TO 1
3260 IF INSTR(1,Z$,IC$(N))>0 THE
N Z2$=IC$(N):Z=N:N=1
3270 NEXT N
3280 IF Z<>-1 THEN LG=1:RETURN
3290 FOR N=4 TO 9
3300 IF INSTR(1,Z$,IB$(N))>0 THE
N Z2$=IB$(N):Z=-N:N=9
3310 NEXT N
3320 IF Z=-1 THEN RF=1 ELSE LG=1
3330 RETURN
```

With a new verb we must obviously have a new action procedure to perform the required task. The listing for THROW is given in listing 8.8d.

Listing 8.8d

```
4000 RF=0:OK=0:ON X+1 GOSUB 4100
,4200,4300,4400,4700,4800,4900,4
500
4499 REM **THROW**
4500 IF PS(Y-5)<1 THEN RF=10:RET
URN
4520 IF Z>0 THEN IF LEFT$(Z2$,1)
<>RM$(PR,1) THEN RF=6:RETURN
4530 IF Z<0 THEN IF LEFT$(Z2$,1)
<>RM$(PR,0) THEN RF=6:RETURN
4540 IF Z1$<>"MAGIC DUST" THEN 4
580
4545 MD=-1
4550 OK=1:PS(Y-5)=PS(Y-5)-1
4560 IF Z>=0 THEN RM$(PR,1)="N":
MN=0 ELSE RM$(PR,0)="N"
4570 RETURN
4580 REM
4670 RF=13:PS(Y-5)=PS(Y-5)-1
4680 IF Z2$="MONSTER" OR Z2$="TR
```

```
OLL" THEN RF=12
4690 RETURN
```

Finally for this section we add listing 8.9d to allow the Dragon to make the appropriate responses.

Listing 8.9d

```
5310 PRINT"YOU THROW ";Z1$;" AT
THE ";Z2$,"HE EATS IT AND LAUGHS
, HA HA":RETURN
5320 PRINT"YOU THROW THE ";Z1$;"
 AT THE",Z2$;", DO YOU FEEL BETT
ER NOW?":FOR N=1 TO 2000:NEXT N:
PRINT"DON'T ANSWER THAT!":RETURN
5380 IF MD=-1 THEN PRINT"THE ";Z
2$;" DISAPPEARS!":MD=0
5410 IF RM$(PR,1)="M" THEN PRINT
"HIDEOUS MONSTER"
```

OK, try the game again, and try throwing objects other than magic dust. Also, try throwing things at objects other than monsters.

CRYSTALS AND SHIMMERING CURTAINS

Our final feature is the introduction of shimmering curtains of impassable energy.

> Some of the doorways in the cave are blocked by curtains of shimmering energy. These can be neutralised by magic dust, but you may need to save that for the monsters. Instead, try to find a crystal, which will act as a key and totally remove the curtain, leaving an open doorway.

No prizes by now for guessing that the first thing we do is add the new words to the inventory. This is done in listing 8.2e.

Listing 8.2e

```
1230 FOR N=0 TO 10
1260 FOR N=0 TO 2
1300 DATA GO,0123,TAKE,56789A,OP
```

```
EN,4,EAT,7689A,RUN,,FIGHT,,TELEP
ORT,,THROW,56789A
1310 DATA NORTH,SOUTH,EAST,WEST,
DOOR,KEY,EMERALDS,FOOD,BATTERIES
,MAGIC DUST,CRYSTAL
1320 DATA TROLL,MONSTER,SHIMMERI
NG CURTAIN
```

I will leave it up to you t o decide what you're going to add in the instructions and we will move straight on to the "create maze" routine. In this section we must make the computer deposit articles around the dungeon, which we do by adding listing 8.4e.

Listing 8.4e

```
1450 IF CRYSTAL>0 THEN L$=L$+"S"
1475 IF O$="S" THEN CRYSTAL=CRYS
TAL-1
1560 IF RND(0)>.6 THEN L$=L$+"C"
1620 IF O$="C" THEN CRYSTAL=CRYS
TAL+1
1800 OB$="NKBN":L$="ODSW":H$="NT
MN"
```

Enter listing 8.5e to describe these new occurences when w e meet them.

Listing 8.5e

```
2030 IF RM$(PR,0)="C" THEN PRINT
 TAB(9);"A SHINY CRYSTAL"
2130 IF RM$(PR,N)="S" THEN PRINT
"THERE IS A SHIMMERING CURTAIN O
FENERGY TO THE ";IB$(N-2)
```

The next listing copes with the existence o f a new object that could be the target of THROW.

Listing 8.7e

```
3250 FOR N=0 TO 2
3260 IF INSTR(1,Z$,IC$(N))>0 THE
N Z2$=IC$(N):Z=N:N=2
```

```
329Ø FOR N=4 TO 1Ø
33ØØ IF INSTR(1,Z$,IB$(N))>Ø THE
N Z2$=IB$(N):Z=-N:N=1Ø
```

With all these options for THROW the procedure to check this is getting a little complicated, but don't worry about this, we will explain it all later. For the moment just add the next listing to your program.

Listing 8.8e

```
452Ø IF Z>Ø THEN IF LEFT$(Z2$,1)
<>RM$(PR,1) AND Z2$<>"SHIMMERING
 CURTAIN" THEN RF=6:RETURN
4545 MD=-1:IF Z2$="SHIMMERING CU
RTAIN" THEN 46ØØ
458Ø IF Z1$<>"CRYSTAL" THEN 467Ø
459Ø IF Z2$<>"SHIMMERING CURTAIN
" THEN 467Ø
46ØØ DR=Ø:FOR N=Ø TO 3
461Ø IF DR=Ø THEN IF INSTR(1,Z$,
IB$(N))>Ø THEN DR=N+2:N=3
462Ø NEXT N
463Ø IF DR=Ø THEN RF=14:RETURN
464Ø IF RM$(PR,DR)<>"S" THEN RF=
15:RETURN
465Ø RM$(PR,DR)="O":R=PR+FNR(DR)
:DR=FND(DR):RM$(R,DR)="O"
466Ø OK=1:PS(Y-5)=PS(Y-5)-1:RETU
RN
```

We now add just one more response to section 9 of the program and that's it.

Listing 8.9e

```
534Ø PRINT"THERE IS NO SHIMMERIN
G CURTAIN  TO THE ";IB$(DR-2):RE
TURN
```

We hope you enjoy the challenge of surviving and beating this final version of our adventure game.

CHAPTER 9

Adventures – A Detailed Look

The previous chapter explained in a general way how the game worked by explaining what each routine did. In this chapter we will take a closer look at how each of the subroutines works, and by the end of the chapter you should be well prepared for inventing your own features to the game, or even for writing your own game completely.

Before we get into the routines themselves, we shall have to look at the DATA, or knowledge, upon which they work. This is stored in the inventories, which represent the computer's knowledge of the world of the dungeons.

The inventories are made up of three string arrays IA$, IB$ and IC$. IA$ is a two DIMensional array of strings. In one DIMension is a list of all the verbs which the program will recognise, and in each corresponding element in the other DIMension there is a list of numbers. This can be seen in figure 9.1. The numbers in IA$ are actually the subscript numbers of the words in IB$. So if we take the verb GO, for example, it has an associated list of numbers 0123, and if we look at IB$, to IB$(3) we see that they contain the words NORTH, SOUTH, EAST and WEST. You may be wondering what the letter A is doing in the list of numbers. In fact we are using it to represent the number 10, so that the computer doesn't get mixed up over 10 being 1 and 0. We will see how this works later.

Figure 9.1 IA$ Array

	0	1
0	GO	0123
1	TAKE	56789A
2	OPEN	4
3	EAT	56789A
4	RUN	
5	FIGHT	
6	TELEPORT	
7	THROW	56789A

Figure 9.2 IB$ Array

0	NORTH
1	SOUTH
2	EAST
3	WEST
4	DOOR
5	KEY
6	EMERALDS
7	FOOD
8	BATTERIES
9	MAGIC DUST
10	CRYSTAL

By looking at these arrays we can see that the player may GO to the NORTH, SOUTH, EAST or WEST, but he may only OPEN a DOOR. He may EAT a whole range of objects (although anything but food will choke him, as we will see later).

When we look at the routine that checks the legality of a player's INPUT we will see how it makes use of the information being stored in this way. The third array IC$ is very similar to IB$. It contains a list of objects, but these are objects which cannot be refered to directly by any of the verbs in IA$. They are referred to indirectly and, in our game, by only one verb – THROW. IC$ is shown in figure 9.3.

Figure 9.3 IC$ Array

1 TROLL
2 MONSTER
3 SHIMMERING CURTAIN

Another "data structure" (to use the lingo) which we must look at before going on to examine the various subroutines is the array which stores the information about the maze itself. The game assumes a maze of 36 rooms/caves/dungeons which are arranged in a 6 x 6 block. If we think about what we need to know about each room, we find that there are six essential pieces of information.

1. Is there a "friendly" object?
2. Is there a "hostile" object?
3. What kind of doorway (if any) is to the NORTH?
4. What kind of doorway (if any) is to the SOUTH?
5. What kind of doorway (if any) is to the EAST?
6. What kind of doorway (if any) is to the WEST?

This means that we need a 36 x 6 array, and you can see the statement DIM RM$(35,5) in line 70 of the Control Program. With this array we can think of each room as having a number between 0 and 35, and to see if room number 4 (say) has a friendly object (e.g. a key) we would look at RM$(3,0) and see if it is equal to K. Now all we need is a convention for representing the various objects, and the convention we have chosen is as follows:

In RM$(n,0) – Friendly objects
K = KEY
E = EMERALD CROWN
F = FOOD
B = BATTERIES
M = MAGIC DUST
C = CRYSTAL

In RM$M(n,1) – Hostile objects
T = TROLL
M = MONSTER

In RM$(n,2) to RM$(n,5) – Entranceways

O = OPEN DOORWAY
D = LOCKED DOORWAY
S = SHIMMERING CURTAIN
W = WALL

The array RJ$ is simply filled with adjectives which are then chosen at random to describe a room the player has just entered. The last array we use in the game is the numeric array PS(4). This is used to store each object the player has on his person.

PS(0) = Number of Keys
PS(1) = Number of Emerald Crowns
PS(2) = Number of Food Packs
PS(3) = Number of Piles of Magic Dust
PS(4) = Number of Crystals

Now that we know how the information is stored we can look at how the various subroutines make use of it.

The first subroutine assigns the inventories, that is, it reads all the verbs, etc., from the DATA statements in lines 1300 to 1320 into the arrays IA$, IB$, IC$ and RM$. This should be easy to follow as it is very similar to the way DATA was read in the sections of our arcade game which defined graphics characters.

The next section, the instructions, is also fairly self explanatory. We have already looked at PRINT statements and FOR/NEXT loops, so enough said!

By contrast, the next section, creating the maze, is probably the most complicated routine in the program. Don't turn over, though, it's still quite easy to follow what's going on. It begins by choosing a random position in the maze at which to put the Crown of Emeralds and another random position for the player's starting point. These positions are stored in the variables CE and PR respectively. The next thing it does is to "walk" from the player's room to the Crown Room, hanging various types of doors and depositing various objects on the way, in such a way as to ensure a feasible path exists through the maze.

It does this by first calling at random either a routine to deposit an object or a routine to deposit a hazard. These routines make a

random choice between the various types of object or hazard, and sometimes even deposit a N (which stands for "Nothing"). The routine which deposits objects keeps a record of how many of each type it has deposited. The number of keys deposited, for example, is stored in the variable KEY.

The next thing the routine does is to choose what kind of door to put up. The choice available depends on what objects have been deposited so far. For example, if KEY is equal to one or more the character D (for locked door) will be added to the list of possibilities (stored in L$) – this occurs in line 1440, and the actual choice of door is made in line 1460. If a locked door is chosen we must assume that the player will use up a key in opening the door (remember they always get stuck or broken) and so the routine takes one off the variable KEY.

Well, now that the routine knows what type of (door or shimmering curtain or whatever) it is going to put up, it needs to figure out where to put it. This is done by calculating the difference between the position of the Crown Room (CE) and the current position of the computer on its "walk" (X). This difference is stored in the variable Y. Bearing in mind that we are using a 6 x 6 grid of rooms, it should be clear that if Y is bigger than six then the Crown Room must be in a row of rooms below the computer's current position. Similarly, if Y is less than minus six the Crown Room must be above us somewhere. A look at figure 9.4 should make this clear. A similar bit of reasoning tells us whether the Crown Room is to the left or right once we are on the same row.

Figure 9.4 The Maze

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
20	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

Once the direction has been decided upon the computer puts the chosen door on the chosen wall, "spirits" itself through the door

into the next room and – very important – puts the same type of door on the other side. (It would be a bit silly to have a door that was locked from one side and open from the other.) You can see all this going on in lines 1480 and 1540.

Once the Crown Room has been reached, Crown of Emeralds and a monster guard are deposited in it and the computer then goes through the entire maze room by room, putting up random doorways wherever it finds blank walls (W) and depositing random objects and hazards in empty rooms. This happens in lines 1800 to 1990 – note lines 1840 to 1870 prevent doors to the outside world from being put up.

Well, you can see that building the maze is quite a job, and this fact is reflected in the twelve seconds which it takes the computer (which is no slouch in these matters) to complete the task. It is so unusual in games-type applications to find something that takes the the computer so long to do, that the player might be forgiven for thinking that the computer has "hung up" on him. That is why the message WAIT is PRINTed at the beginning of the routine.

Having created the maze, the control program now moves into the main loop, which will go round until the game is either won or lost. The first section in this loop calls the routine to describe the room the player is currently in. If the player has just entered the room, as opposed to still being there from the last time around, this routine will also give a short description of the room itself. This condition is "flagged" by the variable NR (for New Room). If it is 1 the routine will describe the room, but if it is 0, it will only describe the room's contents. The routine is in lines 2000 to 2190, and the first two lines are the ones which describe a new room. This is done by choosing two adjectives at random from the array AJ$ and assigning them to the string variables A$ and B$, then including these in the PRINT statement in line 2010. The word CAVE or PASSAGE is also chosen randomly.

If you look at line 270 in the control program you will see the statement ON NR+1 GOSUB 2020, 2000. This is how the program decides whether or not to describe the room. The ON – GOSUB statement looks at the number, or expression, following the word ON, and if it is 1, it GOSUBs to the first line number after the word GOSUB, if it is 2, it GOSUBs to the second line number, and so on.

In this case, this means that if NR is 1, then NR+1 will be 2, and it will GOSUB to line 2000, and describe the room. If NR is 0, however, NR+1 will be 1, and so the control program will GOSUB to line 2020, thereby missing out part of the subroutine which describes the room, and going directly to the section which lists the objects present in the room.

This part of the routine is quite simple. It tests RM$(PR,0), i.e. the "friendly object" element of the room the player is in for each type of object that could be there, and if it finds one it PRINTs it out. Then it does exactly the same for RM$(PR,1) – the "hostile" object element. Note the use of the TAB function to line them up neatly. Just to make the description a bit more interesting, we have added a selection of descriptions of keys – rusty, large, golden and wooden, and these are chosen at random IF RM$(PR,0) = "K", indicating the presence of a KEY. Finally the routine checks the four directions NORTH, SOUTH, EAST and WEST (elements 2 to 5 in RM$ second dimension) and PRINTs out whether a door (it doesn't specify whether it is locked or not) or a shimmering curtain is there.

Once the player has been told of his situation, it is time for him to tell the computer what he wants to do. You will notice that before the player makes his INPUT, there are two PRINT statements. The first of these PRINTs a space at the last position on the screen, causing the screen to scroll up one line, which makes space for the user to enter his INPUT. The second PRINT statement merely serves to move the PRINT position (i.e. the place where the next item will be PRINTed) to the beginning of the last but one line on the screen, which is where we want the player's INPUT to start. Then there is the INPUT statement itself – INPUT U$. This makes the computer wait for the player to type in his instructions, which are placed into the string variable U$.

The next routine checks the player's instruction to see if they make sense. It first makes a quick check to see of the player has given up and typed QUIT, since if he has there is no point in going any further. Look at line 3000. There is a word there that you probably haven't seen before – INSTR. It stands for 'in string" and it is one of many functions the Dragon has for handling strings. What this does is to examine the strings in U$, starting at the first

character (because of the 1 inside the brackets) and see if the string QUIT is to be found in it. The result of this function is a number, not a string, and the number is the position i U$ of the string QUIT. So if the player typed "I «« !!!? QUIT", in a fit of frustration, the INSTR function would produce the number 6. If the "target string" (QUIT in this case) is not in the "object string" (U$ in this case) then INSTR returns the number 0. We don't really care where the word QUIT is within U$, only if it is there at all, so we just ask if INSTR returns a number greater than 0, and if so we set the QUIT flag, Q, to 1, and RETURN. When the legal check routine returns to the control program the QUIT flag is tested, and if it has been set, the program jumps to the END.

Usually the player will not have quit, and so the checking routine has to flex its muscles a little harder. The routine finds words within U$ by looking for spaces, so to avoid any confusion the first thing it does is to remove any leading or trailing spaces from U$. (eg. " GO NORTH " would be trimmed down to "GO NORTH". This is done by the use of another three string functions - LEFT$, RIGHT$ and LEN. LEFT$ takes the form LEFT$(X$,N) and returns a string which consists of the first N characters of the string X$. RIGHT$ takes the form RIGHT$(X$,N) and returns a string which consists of the last N characters of the string X$. For the sake of completeness you may as well meet their sister MID$, which takes the form MID$(X$,N,M) and returns a string consisting of M consecutive characters from X$, starting at character number N. Brother LEN takes the form LEN(X$), and does not return a string, but returns a number equal to the number of characters in X$. With this information you should be able to see how lines 3010 and 3020 strip off the leading and trailing spaces.

The next thing the routine does is to split the player's INPUT (U$) into two strings. The first word (which we assume to be the verb) goes into W1$, and the rest of the string, if there is any more, goes into Z$. Then the routine loops through each "verb" element of IR$ checking it against W1$, and as soon as it finds a match, it stores the verb's subscript number in the variable X and exits the loop. Before entering the loop X is set to –1, so if it is still equal to –1 when the loop has finished, it means that W1$ was

not one of the allowed verbs, so we set RF, used to choose an appropriate response to the player's input, and then RETURN.

If X is not equal to –1 we know that we must have found a legal verb, so the next line tests to see if it was one of the verbs RUN, FIGHT, or TELEPORT. Since these are different from the other verbs in that they do not require any noun to make sense, we can just set the flag LG to tell the control program that the player's INPUT was legal, and RETURN. Otherwise, we will need to check that the rest of the player's INPUT contains a word to which that verb can refer. First, we assign L$ as the list of numbers from IR$ which correspond to the verb we have found. Then we set up a loop which changes each of these numbers in turn from a "character" into a number using the function VAL, and then uses INSTR again to see if the object in IB$, pointed to by that number, is in the player's INPUT. If it is, we store the word's subscript number in Y, store the word itself in Z1$, and exit the loop. We set Y to –1 before entering the loop so that on exit we can test it to see if we found a legal word. You can see this in lines 3160 to 3210. You may wonder what the "&H" is for in line 3180. This is so the computer will interpret "A" as "10" when the string is turned into a number with VAL. "&H" stands for Hexadecimal, or Hex for short. It is a different way of counting used in computing, which counts in 16s instead of 10s, and uses the letters A,B,C,D,E and F to represent the numbers 10 ,11,12,13,14 and 15. MID$(L$,N,1) will provide a character which is either 0 – 9, or A, and by putting "&H"+ in front of it we will build a string that reads "&H3", or "&HA", or whatever the number was. You can "add" strings together like this, but you cannot use other mathematical signs with strings, such as "–" or "*".

For most situations, this will be as far as the inventory search goes. Either Y will still be equal to –1, in which case RF will be set appropriately, or the player's INPUT must contain a legal verb/word combination. There is one verb, however, which requires further testing, and that is THROW. If the verb in the player's INPUT is throw, then X will be equal to 7, and this is tested for before setting LG and RETURNing. This is because the verb THROW needs two other words to make sense. So, if we have found a legal word combination we must now search the

remainder of the player's INPUT for the name of an object at which he can throw things. At this point Z$ contains the player's INPUT minus the first word – the verb. If we search it as it stands we will find the name of the object which is being thrown and conclude that there is an object in the INPUT which could be thrown at. For example, if the player's INPUT is "THROW KEY" (a daft thing to do but perfectly legal) then KEY would be correctly identified as the object being thrown, but incorrectly identified as the object being thrown at, when in fact no target has been specified at all and the INPUT is illegal! We get round this by removing the word KEY from Z$ before doing the search for a target. This happens in lines 3230 and 3240.

There are two arrays containing objects which can be thrown at. IC$ is the one most likely to contain the word in the player's INPUT as this is the array with the words TROLL, MONSTER, and SHIMMERING CURTAIN – the objects at which it is sensible to throw things. For this reason, we search IC$ first. Z is used to store the subscript number of the word (if one is found). If a word is not found (Z equal to –1) we then know that we must search IB$ as well. We search IB$ from subscript 4 onwards. (It doesn't make sense to throw something at a NORTH.) If we find a word, instead of storing its subscript number we store its subscript number multiplied by –1. (Still in Z though). This is so that another routine will know whether the object was found IB$ or IC$. Again, if no word is found, the response flag is set accordingly, otherwise the LG flag is set. In either case there is no more searching to be done so we RETURN.

If the INPUT was not legal, LG will be equal to 0 and the control program will move straight on to the response section of the game. If LG was set to 1 then the program needs to look at the INPUT in the context of the player's current situation and carry out the player's instructions, if possible, since a legal INPUT (for example "TAKE THE KEY"), may still be invalid, if there is no key in the room to take. There are clearly many possible situations that need to be tested for by this routine, so we have split it up into smaller routines, each of which has a much smaller number of possibilities to look for. Which of these subroutines is called depends on which verb has been used. The main routine first

resets a flag called OK to 0, and then calls the appropriate subroutine. If all is well this subroutine will set OK to 1, as well as actually carrying out the player's instructions. The first routine then checks to see if all is OK and sets the response flag accordingly. Now we will look at what each of these subroutines does.

The first one is called if the verb was GO. Like the subroutines for TAKE, OPEN and EAT, the first line tests for the presence of a monster or troll, because if one of these beasts is present, he won't let you go anywhere, open a door, or take or eat anything until you have dealt with him. Assuming there is no beasty present, the routine checks to see if a direction was specified. If not the response flag is set accordingly and we RETURN. Otherwise, we set a variable DR for direction and then look in RM$ to see if there is an open door in the specified direction. If there is, we update PR and set the new room flag, otherwise we set the response flag accordingly. In either case we RETURN. PR holds the player's position within the maze – his room number. We update it using a function defined in the initialisation section of the program. We use it many times in the program and it's much easier to say FNR(DR) than to write out the whole expression every time.

The next subroutine is called when the verb is TAKE. It tests whether the object is actually there for the taking, and also tests whether the object in question is a set of BATTERIES, resetting TIMER if it is. Then it updates the possessions array PS() and RETURNs. TIMER is a special variable in the Dragon which is constantly being incremented at a rate of 50 per second. If you set it to 0 and then read it later by assigning it to a variable, you can work out how much time has elapsed since you set it.

The next subroutine is OPEN. It first checks that the player has a key and, if so, goes on to check that a direction was specified (we must know which door he wants to open) and that the door is locked. If all is well then the possession array is updated (i.e. he loses a key), a flag is set (K) so that the response routine will tell the player that he has lost a key, and RM$ is updated, then we RETURN. (Remember that we need to update two rooms when the state of a door changes, and again we use our defined functions.)

The subroutine EAT checks that the player actually has whatever it is he is trying to eat. Then it checks that he is eating food. If not, RF is set to make him choke, otherwise his strength is replenished, and his possessions array updated, before RETURNing.

The RUN routine finds an open door if there is one and puts the player the other side of it, updating PR at the same time. If however there is nothing to run from, or no open door to run through, FR is set.

The FIGHT routine checks that there is something to fight. If so it kills it, i.e. removes it from RM$ and resets the monster flag MN. It also decreases the player's strength by a random amount, and then RETURNs. TELEPORT simply assigns a random number between 0 and 35 to PR and decreases the strength variable by 200 (teleporting is a strenuous activity).

The last subroutine is THROW, and it is the most complex of these subroutines. It begins by checking that the player has the object he is trying to throw. It then tests to see that the specified target is actually in the room. If either of these tests prove negative, RF is set and we RETURN. If we get past these tests we then find out whether the player is throwing MAGIC DUST, a CRYSTAL, or some other object. This information is in Z1$, as a result of the second part of the inventory search. If it is MAGIC DUST then we disappear the target. (Such is the power of MAGIC DUST.) If the target was a shimmering curtain, however, we must first check that the direction was specifed. If the object thrown was a CRYSTAL, then unless the target was a SHIMMERING CURTAIN it will have no effect, as is true of any other object THROWn. All we want to know is whether the object was thrown at a monster or an inanimate object, so that we can set RF to either make the monster laugh or the computer poke fun at the player.

If the last two sections dealt with the "brains" of the program, this section deals with the "mouth". This is the response routine, and although it is the longest routine in the program, it is also one of the simplest. All the work has already been done to decide what the correct response to the player's INPUT should be. There are 20 basic responses and the actual response is chosen by the

setting of RF (the response flag) to a number between 1 and 20. All the response routine now has to do is make use of the ON . GOSUB statement to call a small subroutine which PRINTs the correct response. A full list of responses and their RF numbers is given in Appendix Two.

After PRINTing the response, the routine checks if the player now has the Crown of Emeralds and, if so, the WON flag is set and we RETURN. The LOST flag is also checked at this point, and if it is set, we also RETURN. If the game is neither WON nor LOST, the response routine examines the strength variable ST and makes an appropriate comment on the state of the player's health. Then it reads TIMER, and if time is running out, it PRINTs a warning to that effect.

The last three routines are only called at the end of a game. The first two simply PRINT out a message to commiserate or congratulate the player, depending on whether he won or lost. After this the control program invites the player to play again, and if he accepts, it calls our final routine, which resets the WON and LOST flags, and the possessions array. The control program then loops back to section 4 of the program, which builds another maze, and the whole thing starts over again. If the player declines to play again the control program moves on to its last statement – END.

CHAPTER 10

Some Parting Remarks.

Well by now you've probably learnt quite a lot about writing and altering the sort of games we have discussed in this book. There should be nothing left to hold you back from writing your own games now. You are now in position to write your own BASIC games and save your hard-earned pennies for only the best, top-quality, machine code, commercial games. Before we leave this introductory book behind though it is probably a good idea to have a look at one of the most important and much used BASIC functions in game writing. Random numbers have been used throughout this book and we have never quite got round to looking at them in detail. They are used a lot because they can help you produce unexpected events, and that is far more interesting than having a game that does exactly the same thing all the time. Without random numbers the aliens would always appear in the same place at the top of the screen and the maze in the adventure would be totally predictable. Of course we could write so complex a program that we wouldn't know which one of the many parts of the program we were playing against but this would take up a lot of memory and, more importantly, a lot of effort from the people writing the game.

The BASIC word associated with random numbers is RND with a parameter to suitably modify the range of the output. RND(1) gives us a random number between 0 and 1 and although it can give a value of 0 it cannot reach 1 (the highest it will ever get is about 0.99999999). Look at the range of numbers with this little program:

```
100 PRINT RND(1)
110 GOTO 100
```

You should see many different numbers between 0 and 1 with no discernable pattern. Actually if you carry on long enough the pattern would repeat after 65536 numbers. This is because RND is really only a pseudo-random function. The computer actually

calculates the next random number from the current one by adding a large prime non-divisor of 2 ↑ 32 then reducing this modulo 2 ↑ 32 to give a number between 0 and 65536. The random number returned is either this number interpreted in two's complement notation for an integer (i.e. for RND with a parameter greater than 1) or this number divided by 65536 for RND(1), so it's all very rational really!!

Here is a short program to simulate the throwing of a die:

```
10 REM PROGRAM TO THROW A DIE
20 PRINT RND(6)
30 GOTO 20
```

The above program will give random numbers in the range 1 to 6. Of course if we wanted to simulate two dice being thrown we would have to produce numbers in the range 2 to 12 but we would work it out by calculating two numbers in the range 1 to 6 and adding them together, not by producing a new function call along the lines of INT (RND(1 1)+ 1.)

Well, that's enough about random numbers. If you have spent enough time playing with (and altering) the games in this book you will be feeling ready to use BASIC in the development of your own games and adventures. Who knows, you may even decide to start writing programs to keep track of your bank balance and such like. Whatever you decide to do remember to keep clearly in mind the overall job. Always split your task up into smaller sub-tasks and never try to solve all the details of one sub-task before you even have an idea what the rest of the tasks are going to be. If you approach programming, or even writing a book, in this way, you will find that you can easily get through even the longest of jobs. This sort of approach is known as structured programming and although BASIC is not intrinsically a structured language we can still approach the building of a program in an orderly fashion. It would almost certainly be worth your while getting a book on structured BASIC programming to help you to expand your knowledge of working BASIC. Remember to go for one that has plenty of programming examples and if possible one with exercises that you can work through. If you got on well using the BASIC blocks in this book then maybe you should consider a career in computing!!

APPENDIX ONE

Arcade Game Variables

ALIENS	Number of aliens left.
AMMO	Number of bullets, etc., left.
AS	ASCII code of character to be PUT by PUT STRING routine.
AX	Current x co-ordinate of alien.
AY	Current y co-ordinate of alien.
BX	x co-ordinate of bomb/fireball.
BY	y co-ordinate of bomb/fireball.
CD	Value to be POKEd to the screen in character definition routine.
CH	Number of character being defined in any batch.
D	Flag to enable downward movement of player.
DD	Flag to indicate dead alien.
F	Flag to indicate firing game.
FIN	Flag to indicate end of game situation.
FUEL	Amount of fuel left.
GM	Number of games played.
H	Flag to indicate hyperdrive enabled.
HIT	Flag to indicate alien hit, or player crashed.
K	Flag to indicate shot has been fired.
L	Flag to indicate left movement enabled.
N	Number of characters to be defined in a given batch.
NV	Inverse flag. If set (1) reverses foreground/ background.
P	Number of aliens that have got past.
PG	Page number of first visible graphics page in character definition routine.
PX	Current x co-ordinate of player.
PY	Current y co-ordinate of player.
Q	Control variable in background loop.
R	Flag to indicate right movement emabled.
RN	Row number of character being defined.
SCORE	Player's score.

ST	First address to be POKEd in character definition routine.
U	Flag to indicate up movement enabled.
X2	X co-ordinate of background object.
XA	Old x co-ordinate of alien.
XG	X co-ordinate of character to be GOT in PUT STRING.
XP	Old x co-ordinate of player.
XS	X co-ordinate of first character in string to be PUT.
Y	Control variable in character definition loop.
Y2	Y co-ordinate of background object.
YG	Y co-ordinate of character to be GOT in PUT STRING.
YP	Old y co-ordinate of player.
YS	Y co-ordinate of first character in string to be PUT.
A$	Used with INKEY$ to wait for player response.
F$	PLAYed in firing routine.
P$	String passed to PUT STRING for PUTting on screen.

ARRAYS

A(1)	Stores alien character.
AB(1)	Stores alien background.
B(1)	Stores player character.
C(1)	Stores bomb/fireball character.
N(1)	Used for PUTting alphanumerics on to the screen. (And for background in bomb routine.)
PB(1)	Stores player background.
T(1),UK(1)	Store background object characters.

APPENDIX TWO

Adventure Game Variables

FLAGS

K	Set to –1 when a key is used up (in opening a door).
LD	Set to 1 to indicate presences of a locked door.
LOST	Set to 1 when player dies!
MD	Set to –1 when magic dust is thrown.
MN	Set to 1 to indicate presence of monster or troll.

NR	Set to 1 when player changes room.
OK	Set to 1 when program successfully performs player's instructions.
Q	Quit flag. Set to 1 when player quits.
RF	Response flag. Set to a number between 1 and 20 to indicate which of 20 responses is to be used in response routine.
WON	Set to 1 when player takes Crown of Emeralds

NUMERIC VARIABLES

CRYSTAL	Used in computer "walk" through maze. While greater than zero it indicates that more crystals than shimmering curtains have been placed in the maze.
CE	Room number of Crown of Emeralds.
DR	Indicate direction of doorway being referred to. 2 to 5 are NORTH, SOUTH, EAST and WEST.
KEY	Used on maze "walk". While greater than zero it indicates that more keys than locked doors have been placed in the maze.
L	Temporary store of lengths of various strings.
M,N	Control variables for various loops.
PR	Player's current room number.
R	Temporary store for random numbers.
ST	Strength level. Starts at 1000. Player dies of exhaustion at zero.
TM	Time passed (in minutes) since game started or since player last acquired new batteries.
V	Control variable for loop in verb inventory search.
X,Y,Z	Used in inventory searches to store subscript numbers of words found. X and Y are also used in building the maze. X stores the computer's room number during the "walk". Y stores the difference.

STRING VARIABLES

A$,B$	Store adjectives from AJ$(4,1) for room description. A$ is also used with INKEY$ to wait for the player to read the instructions.
H$	Strings of hazards in maze building routine.
L$	Temporary store for various strings.
O$	Stores chosen object, hazard, or door to be placed.

OB$	String of objects in maze building routine.
U$	Player's INPUT.
Z$	Player's INPUT minus verb. (And minus Z1$ in inventory search part three.)
Z1$	Object of verb in player's INPUT.
Z2$	Indirect object of verb "throw".

ARRAYS

PS(4)	Numeric array. Stores quantities of each item in player's possession.
IA$(7,1)	List of legal verbs, and pointers to objects to which verbs may refer directly.
IB$(10)	List of objects to which verbs may refer directly.
IC$(2)	List of objects to which the verb THROW may refer indirectly.
RM$(35,5)	Room array. Stores information about presence of friendly objects, hostile objects and doorways, for each room in the maze.
AJ$(4,1)	List of adjectives, for room description.

RF	RESPONSES
1	WHAT?!
2	I DO NOT KNOW THE VERB . . .
3	YOU CANNOT . . . THE . . .
4	YOU CANNOT GO . . .
5	THE DOOR I S LOCKED.
6	I SEE NO . . . HERE.
7	BUT YOU DO NOT HAVE A KEY.
8	THERE IS NO LOCKED DOOR TO OPEN HERE.
9	Not used. (Reserved for future use).
10	YOU DO NOT HAVE THE . . .
11	YOU EAT THE . . . YOU CHOKE TO DEATH!
12	YOU THROW . . . AT THE . . . HE EATS IT AND LAUGHS. HA HA.
13	YOU THROW . . . AT THE . . . , DO YOU FEEL BETTER NOW? DON'T ANSWER THAT!
14	WHICH DIRECTION?
15	THERE IS NO SHIMMERING CURTAIN TO THE . . .

```
16        YOU . . .(Echo player's INPUT).
17        YOU TRY TO . . . BUT YOU ARE ATTACKED
          BY THE . . .
18        YOU RUN.
19        YOU FIGHT THE BEAST, YOU STAVE IN
          HIS SKULL, HE CRUMBLES TO DUST.
20        YOU CANNOT RUN – THE ENTRANCES
          ARE CLOSED.
```

APPENDIX THREE

ASCII Character Codes

Code	*Character*	*Code*	*Character*
32	space	55	7
33	!	56	8
34	"	57	9
35	#	58	:
36	$	59	;
37	%	60	<
38	&	61	=
39	'	62	>
40	(	63	?
41	)	64	@
42	*	65	A
43	+	66	B
44	,	67	C
45	–(minussign)	68	D
46	.	69	E
47	/	70	F
48	0	71	G
49	1	72	H
50	2	73	I
51	3	74	J
52	4	75	K
53	5	76	L
54	6	77	M

Code	*Character*	*Code*	*Character*	
78	N	104	h	
79	O	105	i	
80	P	106	j	
81	Q	107	k	
82	R	108	l	
83	S	109	m	
84	T	110	n	
85	U	111	o	
86	V	112	p	
87	W	113	q	96 – 127
88	X	114	r	appear on screen as
89	Y	115	s	inverse characters.
90	Z	116	t	
91	[	117	u	
92	/	118	v	
93	]	119	w	
94	↑	120	x	
95	←	121	y	
96	@	122	z	
97	a	123	{	
98	b	124	|	
99	c	125	}	
100	d	126	↑	
101	e	127	←	
102	f			
103	g	128 to 255 are block graphic characters.		

APPENDIX FOUR

Decimal/Binary Conversion Table

DECIMAL	BINARY				
0	00000000	5	00000101	10	00001010
1	00000001	6	00000110	11	00001011
2	00000010	7	00000111	12	00001100
3	00000011	8	00001000	13	00001101
4	00000100	9	00001001	14	00001110

DECIMAL	BINARY				
15	00001111	54	00110110	93	01011101
16	00010000	55	00110111	94	01011110
17	00010001	56	00111000	95	01011111
18	00010010	57	00111001	96	01100000
19	00010011	58	00111010	97	01100001
20	00010100	59	00111011	98	01100010
21	00010101	60	00111100	99	01100011
22	00010110	61	00111101	100	01100100
23	00010111	62	00111110	101	01100101
24	00011000	63	00111111	102	01100110
25	00011001	64	01000000	103	01100111
26	00011010	65	01000001	104	01101000
27	00011011	66	01000010	105	01101001
28	00011100	67	01000011	106	01101010
29	00011101	68	01000100	107	01101011
30	00011110	69	01000101	108	01101100
31	00011111	70	01000110	109	01101101
32	00100000	71	01000111	110	01101110
33	00100001	72	01001000	111	01101111
34	00100010	73	01001001	112	01110000
35	00100011	74	01001010	113	01110001
36	00100100	75	01001011	114	01110010
37	00100101	76	01001100	115	01110011
38	00100110	77	01001101	116	01110100
39	00100111	78	01001110	117	01110101
40	00101000	79	01001111	118	01110110
41	00101001	80	01010000	119	01110111
42	00101010	81	01010001	120	01111000
43	00101011	82	01010010	121	01111001
44	00101100	83	01010011	122	01111010
45	00101101	84	01010100	123	01111011
46	00101110	85	01010101	124	01111100
47	00101111	86	01010110	125	01111101
48	00110000	87	01010111	126	01111110
49	00110001	88	01011000	127	01111111
50	00110010	89	01011001	128	10000000
51	00110011	90	01011010	129	10000001
52	00110100	91	01011011	130	10000010
53	00110101	92	01011100	131	10000011

DECIMAL	BINARY				
132	10000100	171	10101011	210	11010010
133	10000101	172	10101100	211	11010011
134	10000110	173	10101101	212	11010100
135	10000111	174	10101110	213	11010101
136	10001000	175	10101111	214	11010110
137	10001001	176	10110000	215	11010111
138	10001010	177	10110001	216	11011000
139	10001011	178	10110010	217	11011001
140	10001100	179	10110011	218	11011010
141	10001101	180	10110100	219	11011011
142	10001110	181	10110101	220	11011100
143	10001111	182	10110110	221	11011101
144	10010000	183	10110111	222	11011110
145	10010001	184	10111000	223	11011111
146	10010010	185	10111001	224	11100000
147	10010011	186	10111010	225	11100001
148	10010100	187	10111011	226	11100010
149	10010101	188	10111100	227	11100011
150	10010110	189	10111101	228	11100100
151	10010111	190	10111110	229	11100101
152	10011000	191	10111111	230	11100110
153	10011001	192	11000000	231	11100111
154	10011010	193	11000001	232	11101000
155	10011011	194	11000010	233	11101001
156	10011100	195	11000011	234	11101010
157	10011101	196	11000100	235	11101011
158	10011110	197	11000101	236	11101100
159	10011111	198	11000110	237	11101101
160	10100000	199	11000111	238	11101110
161	10100001	200	11001000	239	11101111
162	10100010	201	11001001	240	11110000
163	10100011	202	11001010	241	11110001
164	10100100	203	11001011	242	11110010
165	10100101	204	11001100	243	11110011
166	10100110	205	11001101	244	11110100
167	10100111	206	11001110	245	11110101
168	10101000	207	11001111	246	11110110
169	10101001	208	11010000	247	11110111
170	10101010	209	11010001	248	11111000

DECIMAL	BINARY
249	11111001
250	11111010
251	11111011
252	11111100
253	11111101
254	11111110
255	11111111

Robert Erskine & Humphrey Walwyn with Paul Stanley and Michael Bews
Bumper Book of Programs for the Sinclair ZX Spectrum £4.95

Robert Erskine & Humphrey Walwyn with Paul Stanley and Michael Bews
Bumper Book of Programs for the BBC Micro £4.95

Robert Erskine & Humphrey Walwyn with Paul Stanley and Michael Bews
Bumper Book of Progams for the Dragon 32 £4.95

Robert Erskine & Humphrey Walwyn with Paul Stanley and Michael Bews
Bumper Book of Programs for the Oric 1 £4.95

Ian Adamson
The Definitive Companion to the Oric 1 £4.95

Geoff Wheelwright
The Definitive Companion to the BBC Micro £4.95

Jean Frost
Instant Arcade Games for the Sinclair ZX Spectrum £3.95

Jean Frost
Instant Arcade Games for the BBC Micro £3.95

Jean Frost
Instant Arcade Games for the Dragon 32 £3.95

J. J. Clessa
Micropuzzles £2.95

INSTANT INVADERS ... INSTANT LASERS ... INSTANT SPACESHIPS ... INSTANT GAMES ... INSTANT BASIC!

For the newcomer to computing, Jean Frost's **Instant Arcade Games** will be nothing short of a revelation.

With little or no knowledge of BASIC, you can still take a suite of 'skeleton' programs and create your own arsenal of dynamic and totally unique arcade games.

This is not just another collection of listings, but a library of software that also serves as one of the most accessible introductions to structured programming ever written.

For Dragon users who already write their own software, **Instant Arcade Games** offers an invaluable library of imaginative subroutines and user-defined graphics to enhance the efficency and visual impact of their games programs.

Spaceships, motherships, firing and scoring routines – the book is packed with ready-made modules that can be slotted into virtually any kind of arcade-style program.

0 330 28271 9

U.K. **£3.95**

Instant ARCADE GAMES for the DRAGON 32

Pan PCN