The Electron Gamesmaster

Other Granada books for Electron users

The Electron Programmer
S. M. Gee and Mike James
0 246 12340 0

21 Games for the Electron
Mike James, S. M. Gee and K. Ewbank
0 246 12344 3

Electron Machine Code for Beginners
Ian Sinclair
0 246 12152 1

Take Off With the Electron and BBC Micro
Audrey Bishop and Owen Bishop
0 246 12356 7

40 Educational Games for the Electron
Vince Apps
0 246 12404 0

Practical Programs for the Electron
Owen Bishop and Audrey Bishop
0 246 12362 1

Advanced Programming for the Electron
Mike James and S. M. Gee
0 246 12402 4

Adventure Games for the Electron
A. J. Bradbury
0 246 12417 2

Electron Graphics and Sound
Steve Money
0 246 1241 I 3

The Electron Gamesmaster

Kay Ewbank, Mike James and S. M. Gee

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
Distributed in the United States of America
by Sheridan House. Inc

British library Cataloguing in Publication Data

Ewbank, K.
The Electron gamesmaster.
1. Computer games 2. Electron Microcomputer
–Programming
I. Title II. James, Mike III. Gee, S. M.
794.8'028'5404 GV1469.2

ISBN 0–246–12514–4

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

DIGITALLY REMASTERED ON ACORN RISC OS COMPUTERS,
April 2011

Contents

Preface

Mastering programming, like any other skill, takes practice. But it isn't just a case of working at set exercises – at some point you have to start From scratch and implement your own ideas. Many beginners see this as a daunting prospect and would find it much easier if they could get somebody to help by teaching them their craft. This book is intended to provide the same sort of help that an experienced programmer would be able to offer to a relative beginner. It takes you through the stages of program design and implementation pointing out problems and pitfalls, and offering tips and solutions.

The programs developed in the book are all animated graphics games. Some of them at least are familiar classic arcade games. You may have thought that programming such games was beyond your reach but with the help of this book you will soon be writing games even of this level of sophistication and complexity.

The programs are constructed as a series of modules and we suggest you type them in a module at a time, as they are presented. Not only does this lessen the chore of typing in long programs, it also gives you more chance to understand the program in fine detail. Once you've typed in the games we hope that you will enjoy playing them, be confident to tinker with them to modify and improve on them, and be encouraged to write complete programs of your own.

Our thanks are due to Richard Miles and Prue Harrison of Granada, without whose help this book would not have been possible.

Kay Ewbank, Mike James and S. M. Gee

Chapter One
The Sorcerer's Apprentice

This book is rather different from any you may have read about computer programming in the past. It is not just a collection of games that are great fun to play, nor is it another 'learn to program' book. What the Electron Gamesmaster does is to bring together the enjoyment of creating and playing a game with the opportunity to gain that valuable type of knowledge that only comes from practical experience. Today, more people know a programming language than ever before – but knowing a language is only the start of the task of learning how to use it

Many people find that after they have learned BASIC they still have trouble in writing programs that do very much. The types of difficulty that are encountered range from not being able to make a start on the program, through not being able to solve problems that crop up during the program development, to not being able to produce a finished, usable program. In other words, the snags may arise at the start, in the middle or at the end of program development or, of course, in any combination of the three! The point is that, at least at first, it takes a great deal of energy to write a large program. Small programs may be written, tested and discarded in an evening but large programs can take months to complete. The difficulties that arise in working with a large number of lines of BASIC over a long period of time really are new and quite different from anything 'introductions to BASIC' tell you about. When it comes to tackling real problems with BASIC it is not enough In know, for example, what the IF statement does; it is essential to know how to use it in combination with the rest of BASIC to produce a solution in the form of a program that not only works but is a pleasure to use.

The Electron Gamesmaster aims to deal with the problems of writing large programs by explaining how a number of large games programs were written. The reason for the emphasis on games is simply that their appeal is (almost) universal and they contain most of the problems

encountered in other application areas. Where else would you need to combine graphics techniques, sound effects, complicated arithmetic expressions, random numbers, and even Boolean logic, in a single program? A second reason is that games are fun, and owning and using a home computer should be fun! If you want simply to use the book as a collection of games then there is nothing stopping you just turn to the complete listings given at the end of each chapter and type them in but you should find that your enjoyment of the game is greater if you follow the discussion in the chapter and type in the program in smaller chunks.

How to write programs

There is no denying the fact that the best way to learn to write substantial programs is to work with an experienced programmer. In this sense programming is no different from any other craft – you need to serve an apprenticeship before graduating to become a master of the craft. The trouble is that just at the moment, master programmers are in short supply! If you can find someone to help, then by all means take the opportunity to learn at first hand this is always the best way. But whether or not you find such help, The Electron Gamesmaster will be of value to you in improving your programming.

Each chapter starts off with a description of the game that the program is going to implement, pointing out some of the potential difficulties and suggesting methods that are likely to be useful. After reading this introduction you should have the same information that an experienced programmer would have before starting to solve the unexpected problems that crop up during the program development.

Following this introduction the program is presented section by section. Each section of program performs a single action within the program and this is explained in the accompanying text. We suggest that you type in the program as each section is presented. Not only is this a less taxing way of typing in the entire program but it also provides the opportunity to look closely at each section.

Towards the end of each chapter you will find a discussion of what has been achieved in the first version of the program and how it might be improved. Then, if necessary to make the game enjoyable to play, modifications are given and the final version of the program is listed. The complete listing of the program should be used to check that you have typed everything, including the modifications, correctly. While on the subject of accuracy, it is worth saying that all the programs in this book have been printed from working versions without an intervening

(and error-prone) stage of typesetting – that is, they are all presented in the form of of printer listings.

The explanations of how the programs work and why things were done in a particular way should help you to understand the process of writing large programs, but for reasons of length it is not possible to show all the stages that the programs went through during development. However, the programs as listed have not been 'polished' by, for example, renumbering. So, while they reflect the sort of finished product that would be acceptable for use, they still preserve the evidence of programming problems! There is a tendency with published programs either to polish them to the point where they become clinically clean and betray no trace of the difficulties that the programmmer had, or simply to leave them as complex tangles that might be usable but are impossible to understand or modify. The programs in The Electron Gamesmaster haven't been polished at all, so you can see where lines were inserted into the program at a later stage by the odd line numbering. For example, if you find a line number sequence 10,20,25,30,40... then it is a good bet that the programmer had to go back and insert line 25 to take account of something unexpected. On the other hand, all the programs have been written using a programming method that tends to produce programs that are easy to understand.

As you type in and read about each section of the program you should not be afraid to try to test your understanding by running the partially complete program. Of course, unless you add temporary lines of your own you will get error messages from the partial program. However, in most cases you will see something of interest that you should be able to explain in terms of what you already know about the program and this should increase your confidence and make you think a little harder about the program. The Electron Gamesmaster is a practical book, so you should feel free to modify and experiment with the programs both as you go along and once they are complete. Of course even the term 'complete' is relative – it is an old programming saying that no program is ever finished, it just reaches a point where it becomes usable! To encourage you to modify and add to the programs, suggestions concerning likely improvements and even new versions of the game are given at the end of most chapters.

Although this is a practical book, even the most practical subjects need some theory and so the last part of this chapter describes some of the ideas that lie behind programming. These ideas are dealt with at greater length in Advanced Programming for the Electron by Mike James and S. M. (Granada, 1984) and so if you feel they are familiar

then you might like to skip to the start of the next chapter – but a refresher is always useful! On the other hand, if you find that the ideas presented are new to you or you would like to know more about them, then you will find that book is a useful companion to this one.

The need for method

As already mentioned, there is a real difference between writing small and large programs. A small program can be written in one go and most of it can be 'held' in the head of the programmer. However, a large program takes too long to write at one go and has so many lines of BASIC that the programmer who can remember it all is the rare exception. It is possible to write large programs in the same way that small programs are written, but this is very hard work and is one of the reasons why many programmers give up lengthy projects and produce very few finished programs. Even if the necessary amount of time and effort is spent on writing a large program the hard way, the finished product is usually such a mess that any modifications to improve it or in an attempt to fix bugs are often impossibly difficult.

This situation is a great pity because it represents many hours of unnecessarily wasted programming effort. The truth is that if you use a programmming rnethod to organise your effort then writing a large program is no more difficult than writing a short program. Often the suggestion of using a programming method strikes fear into the hearts of freedom-loving programmers. The desire to be unrestricted in the way you write a program is especially understandable if you program for fun. After all, what is the point of turning a hobby into a regimented and routine activity that looks like work? The answer is that a programming method is not a recipe for regimentation, it is simply the art of programming!

Stepwise refinement and procedures

Stepwise refinement is not only a programming method, it is a general strategy for solving problems of all kinds and so it is well worth knowing about. Put simply, stepwise refinement is based on the old idea of divide and conquer. A large program can best be tackled by dividing it into a collection of smaller programs. The smaller programs can then be divided still further and so on until the result is something

small enough to be treated in one go. Described like this, stepwise refinement seems obvious but there is still the problem of how to go about dividing the large program. This is something that is demonstrated by each of the games programs in the rest of this book!

However it is possible in say a little more than this about the theory of stepwise refinement. Suppose you want to write a program that plays chess. The biggest problem that most programmers would have is actually making a start. If you think about it for a moment you should be able to see that any program that plays chess first has to draw the board and initialise everything and then offer the human player a move, then make a move and so on until checkmate. Or, in BASIC:

```
10 PROCdraw_board
20 REPEAT
30 PROChuman_move
40 PROCmachine_move
50 UNTIL checkmate
60 END
```

This is in fact the first stage on the road to a full chess playing program. In other words it is the first stage in refinement of the program. The next stage consists of writing the procedures used in the above section of program (usually called the main program). Each of the procedures should be written by dividing them down into a fist of even smaller procedure calls and so on until the program is written. Notice that when using stepwise refinement it is always easy to make a start on a program by writing down calls to procedures that are not yet written. It also puts off solving any difficult problems until they really have to be solved. For example, in the chess program there is no need to solve the problem of how the machine should move until quite a late stage in the refinement of the program.

Another advantage of using procedures as part of stepwise refinemnent is that the resulting program is very easy to understand and very easy to change. For example, anyone looking at the main program given above for playing chess would soon see that the human player always moved first. This could easily be changed around by swapping lines 30 and 40. Imagine how hard this simple change would be if the parts of the program that made the moves were not separated from the rest of the program in the form of procedures.

There are so many advantages to using stepwise refinement and procedures that it is impossible to list them all. Many of the advantages will become apparent from the examples in the following chapters but

if you would like to know more about programming methods then see Advanced Programming for the Electron.

Structure and style

Stepwise refinement is a method that helps with the writing of a program by breaking it down into a number of procedure calls. However, there comes a point in the refinement when procedure has to do something other than call other procedures. Stepwise refinement gives no guide as to how BASIC should be used to achieve any particular result. The programming method that does have something to say about such things is called structured programming. Structured programming is essentially a technique for writing clear programs by avoiding tying the flow of control into knots. For example, using the GOTO statement it is possible to jump to any part of the program but if you use the GOTO freely then you will discover that your programs are impossible to understand and debug. It is very difficult of follow what is going on in a program that jumps all over the place and if you cannot follow the order in which statements are carried out then there is plenty of room for bugs to move in. Structured programming makes sure that the flow of control is always easy to follow by only allowing the programmer to use a small number of ways of changing it. The most usual selection of ways of changing the flow of control are the IF statement, the FOR loop and the REPEAT loop, and it is possible to write any program using just these three. In other words it is never necessary to use the GOTO statement within a program and this has resulted in structured programming being incorrectly referred to as 'the art of not using the GOTO'. In practice the GOTO is best regarded as dangerous but not prohibited. As long as a section of program is clear, in terms of which statements are executed when, then everything is fine and sometimes the best way to ensure program clarity is to use the GOTO statement.

Another element of program clarity is the use of variable and procedure names that mean something. For example, if a procedure prints the board that a game is played on then it should be called something like PROCprint_board rather than PROCxy4. If you can think up appropriate names for variables and procedures then you should find that your programs are easier to understand and are almost self documenting that is, you can do without REM statements. In practice it is usually very difficult to invent the necessary number of names and keep them short, though there is nothing worse than having

to type out an excessively long name over and over again in a program! Naming variables and procedures is something that you are either good or (like most programmers) bad at, but it is definitely worth the effort.

All this talk of program clarity and style may be worrying if you have always been taught that the nmst important aspect of programming was efficiency – that is, maximising speed and minimising memory usage. It is indeed the case that in any practical program, efficiency is important and occasionally style has to take second place. For example, it is often necessary to use integer variables even though this makes the program look very messy, with percentage signs following following every variable. However, if you produce a fast program, with plenty of memory left over for additions and modifications, and yet which no-one can understand (including yourself after a short period of time), then the additions and modifications will never be made and any bugs are likely to remain. The emphasis should always be on a clear program; anything that has to be done to increase speed or reduce memory usage should be done as modifications to a well written program.

The most obvious way of increasing the efficiency of any program is to use assembler. The Electron is particularly well suited to the mixing of BASIC and and assembler and where necessary in the following chapters this is what is done to improve the efficiency of the games. It is important to realise that the same considerations of programming style apply to assembler as to BASIC. For example, you can use both stepwise refinement and structured programming in assembler in the same way that you would in BASIC. Assembly language is best added to a program by re-writing an existing BASIC procedure in assembler. In other words, first get the program working using nothing but BASIC then identify which procedures within the program are making it slow or take too much space and then re-write these in assembler. If you write procedures in assembler by converting them from BASIC then at least you know that your overall algorithm is correct (it just runs a little slow) and any problems with the assembly language version must be due to an incorrect translation which is much easier to deal with. The rule is that assembly language should only be used where it is absolutely necessary and using it to develop new algorithms is wasteful of programming effort in the extreme. However, the best way to explain these ideas is through the examples that are part of the programs in the following chapters.

Games techniques

Finally, before getting started on the first program in this book, it is worth giving an overview of what each of the games is about and what techniques they illustrate. The game in Chapter Two, Ant Hill, is a simple example of one-dimensional animation and serves to introduce sprites and the animation loop. It is also our first example of using assembly language within a BASIC program. Chapter Three, which presents a game called Leap Frog, moves on to full two-dimensional animation and discusses controlling an object bouncing around the screen, allowing for the effect of gravity. Chapter Four uses an entirely different type of animation, scrolling animation, to produce Frogling, a version of the well known Frogger arcade game. Chapters Five and Six are about one of the most interesting large objects that can easily be animated the snake. Chapter Five develops the basic Snake game, deals with the problems of animating a snake efficiently and introduces the idea of a queue. Chapter Six extends the game to include a number of additional sprites and the result is known as Tadpole. To make things work fast enough Chapter Six introduces an extensive assembly language subroutine that effectively replaces one of the BASIC procedures within the game. Chapter Seven takes up a traditional game, Snakes and Ladders, and implements an up-to-date computer version of it. The main technique explained as part of Snakes and Ladders is the way that a sprite can be animated against a complicated background without destroying the background each time it moves.

Each of the games introduces something new in terms of animation techniques and in the use of your computer and the final chapter draws these together with comments on designing and implementing games in general and on using the Electron in particular. By the time you reach it you should have learned a great deal about both subjects and had plenty of fun along the way.

Chapter Two
Ant Hill

Ant Hill is a simple but effective program that involves the animation of a number of objects. The basic idea of the game is to guide a man through tunnels that belong to an ant colony with the aim of reaching and destroying the nest of eggs located at the deepest point. The difficulty of this task is increased by having to work within a time limit and by having to avoid soldier ants positioned at each level of the tunnels. Thee major problem in implementing a game of this sort lies in animating a number of objects, the man and all the soldier ants, at the same time. As well as dealing with this particular problem this chapter also develops some of the standard methods that will be used without further comment in subsequent chapters.

The game design

Before starting to write any program it is a good idea to try to specify, in as much detail as possible, what it should do. Games programs are slightly different from other applications in that it is usually not possible to give an accurate outline of the final game before at least part of it is implemented. The reason for this is that it is very difficult to predict how elements of a game will work without trying them out. Even after writing a large number of games programs it is still difficult to predict the overall effect of combining different elements from existing games to produce a new one. However, it is still worth working out what you expect a game to do before you start writing any of it, for the simple reason that it gives you time to get any major changes out of your system!

The design of Ant Hill (and many other dynamic games) falls nuturally into three parts:

1. the background graphics

2. the moving characters and the rules of movement
3. the consequences of winning and losing

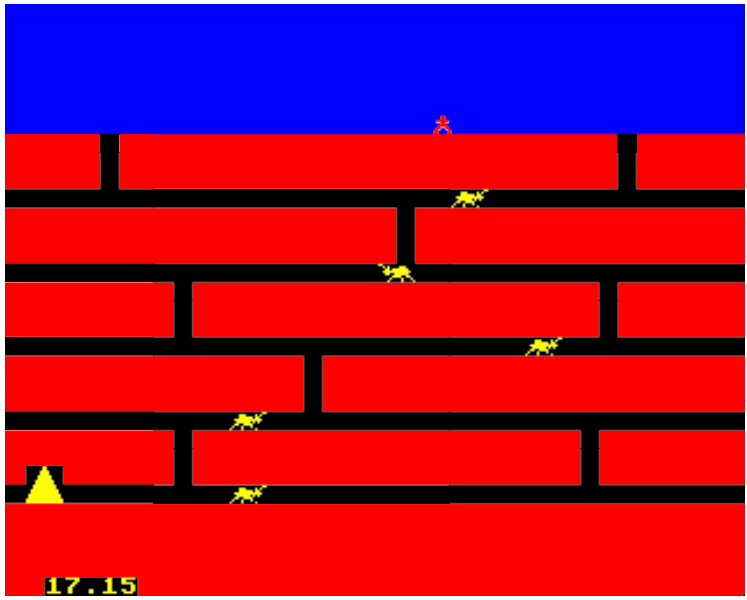


Fig. 2.1.

The background of Ant Hill consists of a number of horizontal tunnels connected by vertical shafts (see Fig. 2.1). The reason for using a mixture of one and two shafts to connect the tunnels is that it promises to give the game more variety of strategy. When only one shaft connects two levels then there is only one route available and playing the game becomes a matter of timing, but when there are two shafts the player has the opportunity of choosing which one to go down. The general layout of the background suggests that at least three colours are going to be needed one for the tunnels, probably black, one for the earth, probably red and one for the sky above, almost certainly blue. This suggests that either a four- or a sixteen-colour mode needs to be used. A choice of the sixteen-colour mode would, however, restrict us to a horizontal resolution of only 20 printing columns and, as Ant Hill is a game that depends on a great deal of horizontal movement, the four-colour mode seems a better choice.

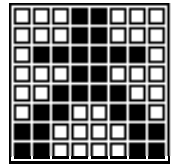
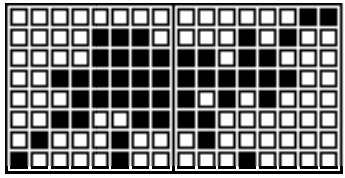


Fig. 2.2. Graphics character for man

The man shape is easily impelemented as a single user-defined graphics character (see Fig. 2.2). Using a single character means that the tunnels and the shafts that connect them only now have to be one character wide. However, the ant shapes are much more difficult to implement in a single eight by eight dot character because they are fairly long and thin. The solution is to use two user-defined graphics characters (see Fig. 2.3). The fact that the ants only move horizontally along the tunnels, and never up or down the shafts, means that even though they are composed of two characters the tunnels and shafts still only need to be one character wide.



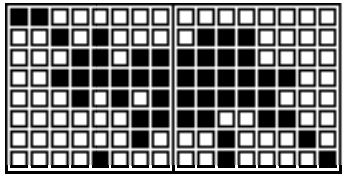


Fig. 2.3. Graphics characters for ants

The rules of movement for the ants are easy to define. Each level will have one ant confined to that level able to :move along the tunnel. If an ant happens to arrive the current of the man character then the

game is over and the man can be taken back to the nest to be eaten! At this stage it is difficult to say exactly how the ants should move to produce a good game but it seems reasonable to start off with the assumption that they should move randomly along the tunnel and occasionally change their direction, also at random. This choice of 'random ants' should provide a sufficient level of difficulty for the player as long as the time limit to complete the course is short enough.

The rules of movement for the man are a little more complicated. Like the ants he should be allowed to move along the tunnels but he is also allowed to move down the shafts. The game would probably be too easy if he was also allowed to move back up the shafts so the player's control over the main is limited to the left and right arrow keys for horizontal motion and the down arrow key for vertical motion down a shaft. If the down arrow key is pressed when the man is over a shaft then he should 'jump' down it to the next level. When the man is not positioned over a shaft the down arrow keys should have no effect. Apart from details such as making sure that neither the ants nor the man can move off the edge of the screen, this completes the description of how they should move.

There are three possible ways in which the game could end:

1. the man reaches the nest
2. an ant captures the man
3. the time limit is exceeded and the eggs hatch

Although there is no doubt that a successful game has to be exciting whilst in progress, there is also a lot of scope for constructing interesting endgames. For example, in this case, when the man reaches the nest he could be rewarded by a fanfare, when an ant captures a man it could drag him down to the nest and when the time limit is reached this could be signalled by an ant population explosion!

After the game specification it is almost time to start writing the program, but first it is worth going over some of the ideas and difficulties of computer animation.

The technique of animation

The computer animation of small shapes is simplicity itself. To make anything appear to move, all you have to do is repeatedly to print it at one location, then erase it (usually by printing a biank) and then reprint it at its new location. If you can repeat the 'print-erase-print' cycle

sufficiently fast you can create the illustion of fairly smooth motion. In practice there are two factors that control how smooth computer-generated motion will appear. The first is the time lapse between erasing and reprinting the shape. This we will call 't1' and it should always be as short as possible because it is the length of time that the shape is not visible on the screen. If t1 is too long then the object will appear to 'twinkle' and even 'flash'. The second factor is the time lapse between each printing of the shape. This we will call 't2' and it can be as long as you like because it controls the speed at which the object appears to move. In most cases we want the object to move quickly and so t2 needs to be short, but this isn't always so. The way that the different time tntervals affect the animation can be seen in Fig. 2.4.

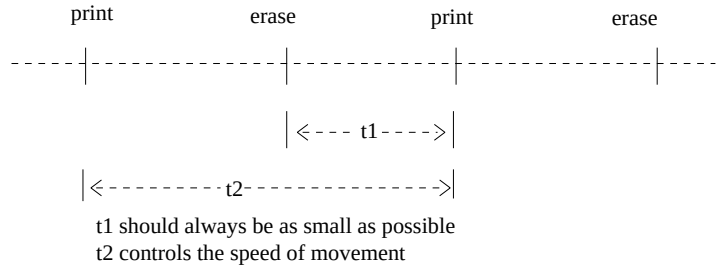


Fig. 2.4. Animation timing.

There is another factor that controls the speed at which something moves that has so far been ignored. It has been assumed that each time the object moves on the screen it moves by the smallest possible amount that is, one character location. Moving the object by one character location each time does have the advantage that the motion appears as smooth as possible and that the object is actually printed at each position that it appears to pass through. On the other hand there is one big advantage to moving the object by more than one character location at a time – speed! It is often possible to make an object appear to be moving quite fast using nothing but BASIC by making it move more than one character location at a time. In Ant Hill both methods of animation will be used: the man will only move one character location at a time but the ants will sometimes move greater distances in one go.

The 'print-erase-print' cycle is the foundation of animated graphics but this still leaves the problem of how to keep track of a number of moving objects within a program. Each object on the screen is associated with five quantities and hence five variables: the character that defines its shape, its current position in terms of both its x and its y screen co-ordinates, and the amounts that its x and its y co-ordinates

should change each time through the animaation loop. These last two quantities can be thought of as velocities because they govern how far the object will move each time through the animation loop. For example, an object‹s shape might be defined by a character stored in C$, its position in X and Y and its velocity in VX and VY. Its animation loop would be:

```
PRINT TAB(X,Y);" ";:REM erase object
X=X+VX:Y=Y+VY      :REM update co-ordinates
PRINT TAB(X,Y);C$; :REM reprint object
```

In practice animation loops are generally a little more complicated because the object's velocity is usually changed as a consequence of where it is on the screen. For example, an animation of a bouncing ball would invoke changing the velocity whenever the ball bounced against the screen boundary.

That is all there is to the theory of simple animation. An object that is animated in the way described is usually referred to as a sprite. The Ant Hill program uses a total of six sprites, five ants and one man, and is a good demonstration of the fact that in computing theory is often no more than a guideline!

The main program

The main program of all dynamic games looks pretty much the same. There is usually some initialisation followed by the animation loop and then routines that deal with the end of the game. Ant Hill is no exception.

```
  10 REM ANT HILL
  20 MODE 1
  30 PROCinitialise
  40 PROCtitle
  50 PROCdraw_scene
  60 REPEAT
  70 PROCone_move
  80 UNTIL TIME_UP OR HIT OR HOME
  90 PROCend_game
 100 CLS
 110 END
```

Lines 20 to 50 are procedure calls that are concerned with getting the game set up. Lines 60 to 80 form the animation loop and lines 90 to 110 deal with the end of the game. PROCinitialise is intended to do such things as set up graphics characters, logical to physical colour

assignments and generally initialise any variables that need initialising! PROCtitle simply prints a title fram and sets the difficultv level of the game. PROCdraw_scene is intended to draw the system of tunnels and shafts ready for the game to sart. You could say that the heart of the program is PROCone_move which is responsible for making one move of both the man and the ants. PROCone_move is repeatedly called until one of the three variables TIME_UP, HIT or HOME become equal to TRUE. These three variables are used to indicate the three possible reasons for the game coming to an end and they are set by tests within PROC_move. Notice that any numeric variables can be used to store either of the two 'truth values', TRUE or FALSE, and so can be used by procedures to 'pass back' the result of a test. All that now remains is to write each of the procedures used by the main program!

PROCinitialise and PROCtitle

The first two procedures, PROCinitialise and PROCtitle, were not difficult to write. PROCinitialise was, however, one of the few procedures in the program that was changed during development by having lines added to it. The reason for this was that it was not at all clear what variables would have to be initialised until other procedures had been written. Rather than present the first version of PROCinitialise and then introduce all the changes as the other procedures are explained, it is less confusing simply to give the final form of the procedure.

```
1000 DEF PROCinitialise
1010 VDU 23,224,&00,&0E,&0F,&3F,&1F,&33,&44,&84
1020 VDU 23,225,&03,&14,&D8,&FC,&A8,&C0,&40,&10
1030 VDU 23,226,&00,&70,&F8,&FC,&F8,&CC,&22,&21
1040 VDU 23,227,&C0,&28,&1B,&3F,&19,&03,&04,&08
1050 VDU 23,228,&18,&18,&7E,&18,&3C,&66,&C3,&C3
1060 VDU 23,229,&01,&01,&03,&03,&07,&07,&0F,&0F
1070 VDU 23,230,&1F,&1F,&3F,&3F,&7F,&7F,&FF,&FF
1080 VDU 23,231,&80,&80,&C0,&C0,&E0,&E0,&F0,&F0
1090 VDU 23,232,&F8,&F8,&FC,&FC,&FE,&FE,&FF,&FF
1100 VDU 19,0,0,0,0,0:REM 0=BLACK
1110 VDU 19,1,4,0,0,0:REM 1=BLUE
1120 VDU 19,2,3,0,0,0:REM 2=YELLOW
1130 VDU 19,3,1,0,0,0:REM 3=RED
1140 XM%=RND(20)+10
1150 YM%=6
1160 DIM A%(4,3),A$(4)
1170 FOR I=0 TO 4
1180 A%(I,1)=RND(30)+5
```

```
1190 A%(I,2)=I*4+10
1200 A%(I,3)=1
1210 A$(I)=CHR$(224)+CHR$(225)
1220 NEXT I
1230 HIT=FALSE
1240 HOME=FALSE
1250 TIME_UP=FALSE
1260 ENDPROC
```

The first part of PROCinitialise creates the necessary user-defined characters. Lines 1010 and 1020 define the two characters that go together to form an ant shape facing right and lines 1030 and 1040 do the same thing for an ant shape facing left. That is, following PROCinitialise,

```
PRINT TAB(X,Y);CHR$(22l);CHR$(225);
```

will print an ant facing right at X,Y and

```
PRINT TAB(X,Y);CHR$(227);CHR$(226);
```

will print an ant facing left at X, Y. The need to use two different ant shapes hasn't been mentioned until now but the fact that the ants move left and right makes it absolutely essential. If only one ant shape was used for both directions there would be times when the ants appeared to walk backwards! You might think that keeping track of which pair of ant characters have to be used for any particular ant would be difficult, but in fact it is fairly easy, as will become apparent when we consider PROCmove_ant.

Line 1050 defines the man character as CHR$(228) and lines 1060 to 1090 define four characters that fit together to make up a pyramid shape that represents the nest. That is:

```
PRINT TAB(X,Y-1);CHR$(229);CHR$(231);
PRINT TAB(X,Y);CHR$(238);CHR$(232);
```

will print the four characters that make up the nest with its bottom left hand corner at X, Y.

Lines 1100 to 1130 determine which physical colour each logical colour code will produce. It is always a good idea to try to define all of the logical colours available in a given mode at the same point in a program and to add REM statements to indicate the colour assignment. This makes it much easier to change the colour selection at a later date if required.

The rest of PROCinitialise is concerned with creating and initialising

variables used later in the program. Lines 1140 and 1150 set the man's initial position in XM% and YM%. His initial vertical position is always fixed at 6 so that he starts the game above ground, but his horizontal position is random. Lines 1160 to 1220 initialise the current positions of the five soldier ants, their direction of travel and shapes (i.e. left- or right-facing). The array A%(4,3) is used to hold most of the information about each ant. A%(1,1) gives the x-co-ordinate of ant 1, A%(1,2) gives the associated y co-ordinate and A%(1,3) gives the ant's direction of travel. If A%(I,3) is 1 then ant I is moving to the right and if it is –1 then it is moving to the left. Finally, the correct pair of characters that form the ant‹s shape are stored in A$(I). Thus to print and I (remember I is in the range 0 to 4) use

```
PRINT TAB(A%(I,1),A%(I,2));A$(I);
```

Line 1180 initialises each ant's horizontal position randomly in the range 6 to 36. Line 1190 initialises each ant's vertical position so that there is one ant per tunnel. That is, ant 0 is printed on line 10, ant 1 on line 14 and ant 2 on line 18 and so on. All the initial ant directions are set to I, making all the ants move to the fight (line 1200) and so line 1210 stores a right-facing ant shape in A$(I) to all right-moving ants. The final part of PROCinitialise sets the variables that are used to indicate the end of the game to FALSE (lines 1230 to 1250).

PROCtitle is simplicity itself! Indeed, writing this procedure simply involves finding a reasonable form of words to explain the game to a new player.

```
8000 DEF PROCtitle
8010 CLS:COLOUR 128+0:COLOUR 2
8020 PRINT TAB(12,3);"A N T   H I L L"
8030 PRINT STRING$(20,A$(1))
8040 PRINT TAB(5,10);"In this game you have a
     fixed"
8050 PRINT "amount of time to destroy the ants
     nest"
8060 PRINT "before the eggs hatch!"
8070 PRINT
8080 PRINT "Use the arrow keys to move but"
8090 PRINT "BEWARE: the soldier ants guarding
     the"
8095 PRINT "tunnels will capture you and take
     you"
8096 PRINT "to the nest to feed the young!"
8100 PRINT
8110 PRINT "Good luck"
8300 PRINT TAB(5,25);"Which difficulty level -"
8305 PRINT
```

```
8310 PRINT "1. Expert, 2. Medium or 3. Novice";
8320 INPUT DF
8330 IF DF<1 OR DF>3 THEN GOTO 8300
8340 MAX=(20+DF*10)*100
8350 VDU 23,1,0;0;0;0;
8360 ENDPROC
```

The only points of note in this procedure are the use of the STRING$ function to print a line of ants as an underlining for the title (line 8030), and the VDU 23 command in line 8350 which turns the text cursor off for the duration of the game. Lines 8300 to 8320 ask the user to specify the difficulty level of the game. Line 8330 checks to make sure that DF is in the correct range and line 8340 sets MAX to the amount of time that the user is allowed at the given difficulty level.

PROCdraw_scene

PROCdraw_scene is the first procedure that actually does very much in the way of graphics. Its job is to draw the system of tunnels and shafts ready for the game to commence. As it will only be used once during the game it is really ‘speed critical’ and our main concern while writing it should be to produce a clear procedure that is easy to modify.

```
2000 DEF PROCdraw_scene
2010 COLOUR 128+3
2020 CLS
2030 COLOUR 128+0
2040 FOR X=0TO 39
2050 PRINT TAB(X,10);" ";
2060 PRINT TAB(X,14);" ";
2070 PRINT TAB(X,18);" ";
2080 PRINT TAB(X,22);" ";
2090 PRINT TAB(X,26);" ";
2100 NEXT X
2120 COLOUR 128+1
2130 FOR X=0 TO 39
2140 FOR Y=0 TO 6
2150 PRINT TAB(X,Y);" ";
2160 NEXT Y
2170 NEXT X
2175 COLOUR 128+0
2180 PROCshafts(2,7,9)
2190 PROCshafts(1,11,13)
2200 PROCshafts(2,15,17)
2220 PROCshafts(1,19,21)
2230 PROCshafts(2,23,25)
```

```
2240 COLOUR 3:COLOUR 128+1
2250 PRINT TAB(XM%,YM%);CHR$(228);
2260 COLOUR 2:COLOUR 128+0
2270 PRINT TAB(1,25);CHR$(229);CHR$(231);
2280 PRINT TAB(1,26);CHR$(230);CHR$(232);
2290 COLOUR 2
2300 FOR I=0 TO 4
2310 PRINT TAB(A%(I,1),A%(I,2));A$(I);
2320 NEXT I
2330 TIME=0
2340 ENDPROC
```

Line 2010 clears the screen to logical colour 3 which is currently set to red Although the COLOUR instruction in line 2010 could be written as

```
COLOUR 131
```

the fact that it sets the background colour to 3 is more obvious if it is written as

```
COLOUR 128+3
```

Lines 2030 to 2100 print the horizontal tunnels in black by first setting the background colour to black (line 2030) and then printing five horizontal lines of spaces. This creates five black bands through the red background. Next, lines 2120 to 2170 print a block of blue spaces to represent the sky. Alter this all that remains is to print the vertical shafts that connect the tunnelsa This is a problem that is best solved by calling another procedure – PROCshafts(NO,Y1,Y2) which will draw NO shafts vertically, each one starting at Y1 and ending at Y2. So, for example, PROCshafts(2,7,9) draws two shafts between the tunnels at Y=6 and Y=10. Lines 2180 to 2230 use the yet-to-be-written PROCshafts to connect the tunnels.

Lines 2240 and 2250 print the man figure at XM%,YM% using red foreground and blue background. At this stage of the game the man is above ground and hence needs to be printed with a blue background but later he will be in shafts and tunnels and the background colour will then need to be black. Lines 2260 to 2280 print the nest at the far left hand end of the bottom tunnel. Finally, lines 2290 to 2320 print the ants and line 2330 zeroes the TIME ready for the game to start.

The only procedure used by PROCdraw_scene is PROCshafts (NO,Y1,Y2) and this has to be completed before moving on to the other procedures:

```
9000 DEF PROCshafts(NO,Y1,Y2)
9010 X1=RND(10)+4:X2=RND(10)+25
9020 FOR Y=Y1 TO Y2
9030 IF NO=1 THEN PRINT TAB((X1+X2)/2,Y);" ";
9040 IF NO=2 THEN PRINT TAB(X1,Y);" ";
     TAB(X2,Y);" ";
9050 NEXT Y
9060 ENDPROC
```

Line 9010 fixes the positions of two shafts at X1 and X2 at random. The random numbers are generated in such a way that the shaft specified by X1 is to the left of the screen and the shaft specified by X2 is to the right of the screen. If two shafts are required then they are both printed by line 9040 within the FOR loop (lines 9020 to 9050). If only one shaft is required then it is printed at the average of Xl and X2, which is a random position tending roughly towards the middle of the screen (line 9030).

PROCone_move – the heart of the game

The most important procedure in the entire game is PROCone_move. As it is going to be called so many times it is important for it and any procedures that it calls to work as quickly as possible. This said. it is still worth trying to keep things clear and simple and to this end the best way to write PROCone_move is as a series of calls to other procedures:

```
4000 DEF PROCone_move
4010 PROCmove_man
4020 PROCmove_ant
4030 PROCtime
4040 ENDPROC
```

The job of PROCmove_man is to allow the player the opportunity of moving the man and to check for the possibility that the move has resulted in the man reaching the nest. Similarly, PROCmove_ant is concerned with moving the ants and checking to see if the man has been captured. PROCtime prints the time since the start of the game and checks to see if the allotted time is up.

Clearly, the way that PROCone_move has been written puts off all the difficult decisions until the other procedures are written but this strategy is intentional. Programming problems should always be solved in small steps. PROCmove_man is the next procedure to be written:

```
5000 DEF PROCmove_man
5010 IF YM%>9 THEN COLOUR 128+0 ELSE
     COLOUR 128+1
5020 COLOUR 3
5030 PRINT TAB(XM%,YM%);" ";
5040 IF INKEY(-26) AND XM%>0 THEN XM%=XM%-1
5050 IF INKEY(-122) AND XM%<39 THEN XM%=XM%+1
5060 IF INKEY(-42) AND FNC(XM%,YM%+1)=0 THEN
     PROCdown_shaft
5070 PRINT TAB(XM%,YM%);CHR$(228);
5080 IF XM%=3 AND YM%=26 THEN HOME=TRUE
5090 ENDPROC
```

Line 5010 checks to see what the vertical positionofthe man is. Ifhe is still above ground the background colour is set to blue and if he is below ground the background colour is set to black. Line 3030 blanks out the man at his current position XM%,YM%.

Lines 5040 to 5060 then check each of the three possible arrow keys in turn. Using INKEY with negative parameter values is the quickest way of finding out if a key is pressed or not and avoids any problems that may arise because of the keyboard buffer storing old key-presses. Line 5040 checks for the left arrow key, line 5050 checks for the right arrow key and line 5060 checks for the down arrow key. The only other point worth mentioning is that in lines 5040 and 5050 also make sure that the attempted move would not take the man off the edge of the screen. Line 5060 calls PROCdown_shaft to make the man jump down a shaft only if the down arrow key is pressed and the character location below the man's current position is black. The function FNC is used to discover the colour of the character location just below the man. In fact FNC works by returning the colour of a point somewhere near the middle of a character location:

```
9100 DEF FNC(X%,Y%)
9110 X%=3+32*X%
9120 Y%=1020-32*Y%
9130 =POINT(X%,Y%)
```

Lines 9110 convert the character co-ordinates given X%,Y% to the co-ordinates of a point roughly in the middle of the character location on the high resolution screen. Then line 9130 uses the POINT function to return its logical colour. PROCmove_man finishes by reprinting the man character at the new position given by XM%,YM% (line 5070) and then checks to see if the nest has been reached (line 5080).

The only procedure left to tackle for PROCmove_man to work is PROCdown_shaft:

```
5500 DEF PROCdown_shaft
5505 COLOUR 128+0
5510 FOR I=1 TO 4
5520 YM%=YM%+1
5530 PRINT TAB(XM%,YM%);CHR$(226);
5540 SOUND 1,-15,148-I%*4,2
5550 IF ADVAL(-6)<>15 THEN GOTO 5550
5560 PRINT TAB(XM%,YM%);" ";
5570 NEXT I
5580 ENDPROC
```

This starts by setting the background colour to black (line 5505) because a shaft is always black! Then lines 5510 to 5570 move the man down the shaft one character location at a time. Each move is accompanied by a sound produced by line 5540. Line 5550 uses ADVAL(6) to wait for the end of the note by testing for the sound queue to be empty (i.e. to have 15 free places).

PROCmove_ant tackles the first real problem that has to be solved. The trouble is that there are one man and five ants. If PROCmove_ant were to move all five ants each time it was called then the man figure would be very sluggish to respond to keypresses from the user. The reason for this is the time it takes to move five ants! The obvious solution is to move only one ant, selected at random, for each possible move of the man. This is exactly what PROCmove_ant does:

```
3000 DEF PROCmove_ant
3005 COLOUR 128+0:COLOUR 2
3010 Q%=RND(5)-1
3020 PRINT TAB(A%(Q%,1),A%(Q%,2));SPC(2);
3030 IF RND(1)<.01 THEN PROCreverse(Q%)
3040 A%(Q%,1)=A%(Q%,1)+A%(Q%,3)*RND(3)
3050 IF A%(Q%,1)<3 THEN A%(Q%,1)=3:
     PROCreverse(Q%)
3060 IF A%(Q%,1)>36 THEN A%(Q%,1)=36:
     PROCreverse(Q%)
3070 PRINT TAB(A%(Q%,1),A%(Q%,2));A$(Q%);
3080 HIT=FNgot_man(Q%)
3090 ENDPROC
```

Line 3010 sets Q%, randomly to a number between 0 and 4 to determine which ant will be moved. Then line 3020 erases the ant from its current position. Line 3030 calls PROCreverse at random to reverse the direction of ant Q%. RND(1) will be smaller than .01 on average once in 100 calls to PROCmove_ant. Line 3040 increases or decreases the ant's x co-ordinate depending on the value stores in A%(Q%,3). If A%(Q%,3) is +1 then RND(3) is added to the x co-ordinate, thus moving the ant to the right; if A%(Q%,3) is −1 then RND(3) is

subtracted from the x co-ordinate, thus moving the ant to the left. The amount that the ant moves is given by the value of RND(3). This means that the ant can move by 1, 2 or 3 character locations at a time. Although in general moving by anything than one character location at a time is a potential source of trouble, this is really the only way in which the impression that the ants move quickly can be achieved. After updating the ant's x co-ordinate it is checked by lines 1050 and 3060 to see if the ant is about to go off either edge of the screen. If so, PROCreverse(Q%) is called to change the ant's direction. Finally line 3070 reprints the ant at its new position and line 3080 used FNgot_man(Q%) to check whether the ant just moved has captured the man not.

PROCreverse(Q%) is the only procedure used by PROCmove_ant. All it has to do is change the direction of motion of the ant and the pair of graphics characters.

```
3500 DEF PROCreverse(Q%)
3510 IF A$(Q%)=CHR$(224)+CHR$(225) THEN
     A$(Q%)=CHR$(227)+CHR$(226) ELSE
     A$(Q%)=CHR$(224)+CHR$(225)
3520 A%(Q%,3)=-A%(Q%,3)
3530 ENDPROC
```

The function FNgot_man is also easy to write:

```
9200 DEF FNgot_man(Q%)
9210 IF A%(Q%,2)<>YM% THEN =FALSE
9220 IF A%(Q%,1)<>XM% AND A%(Q%,1)+1<>XM%
     THEN =FALSE
9230 =TRUE
```

The only remaining procedure used by PROCone_move is PROCtime:

```
4500 DEF PROCtime
4510 PRINT TAB(2,31);TIME/100;
4520 IF TIME>MAX THEN TIME_UP=TRUE
4530 ENDPROC
```

Line 4510 prints the current time and line 4520 tests to see if the allotted time is up and the game is at an end.

Now that PROCone_move and all the procedures and functions that it uses have been defined, the game can be tried out apart from the endgame routines. In this part of the program, integer variables have been used to give the maximum speed of execution. However, even with this care the Electron still only runs the program at a barely acceptable speed.

PROCend_game

The choice of how to finish a game is often something that is limited by the amount of time already spent on getting the main parts of the game working. There are three possible conclusions to Ant Hill, and PROCend_game has to identify each and call appropriate procedures:

```
6000 DEF PROCend_game
6010 IF HIT THEN PROCcapture:GOTO 6100
6020 IF TIME_UP THEN PROChatch:GOTO 6100
6030 IF HOME THEN PROCdestroy_nest
6100 VDU 23,1,1;0;0;0;
6110 PRINT TAB(2,29);"Another game ";:INPUT A$
6120 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 6100
6130 IF LEFT$(A$,1)="Y" THEN RUN
6140 ENDPROC
```

PROCcapture is called if an ant has reached the same screen location as the man. PROChatch is called if the garneends due to the time limit being reached and PROCdestroy_nest is called if the man reaches the nest within the time limit. Following a call to one of these procedures line 6100 restores the text cursor and lines 6110 to 6130 ask the player if he wants another game.

Of the three procedures called by PROCend game, PROCcapture is seemingly the most complicated and so a description of this is left until last. PROChatch simply prints ants at random positions tending toward the centre of the screen, accompanied by clicking noises:

```
6500 DEF PROChatch
6510 PRINT TAB(2,30);"THE ANTS ARE HATCHING "
6520 FOR I=1 TO RND(20)+20
6530 X=RND(25)+5:Y=RND(10)+10
6540 PRINT TAB(X,Y);A$(RND(5)-1);
6550 SOUND 0,-15,RND(4)+3,RND(2)
6560 NEXT I
6570 ENDPROC
```

PROCdestroy_nest rewards the player with a simple fanfare:

```
6800 DEF PROCdestroy_nest
6810 PRINT TAB(2,30);"YOU DID IT!"
6820 DATA 72,.5,80,.5,88,.5,96,.5,100,1,100,1
6840 ENDPROC
6890 DATA 999,999
6900 READ P,D
```

```
6910 IF P=999 THEN GOTO 6940
6920 SOUND 1,-15,P,D*10
6925 SOUHD 1,0,0,1
6930 GOTO 6900
6940 ENDPROC
```

The notes of the fanfare are specified by the values in the DATA statements (lines 6830 and 6890). Each note corresponds to a pair of values, the first giving the pitch and the second the duration. A pitch of 999 is used to mark the end of the sequence of notes and this is detected by line 6910

The idea implemented by PROCcapture is that the ant that captures the man should 'drag' him to the nearest shaft and drop him down to be collected by the ant at the next level. This is repeated until the ant at the lowest level drags him to the nest, when the game is finally over. This sounds like quite a cmnplicated sequence of events but in practice it is not so difficult.

```
6200 DEF PROCcapture
6210 PRINT TAB(2,30);"You are captured!"
6215 IF Q%=4 THEN GOTO 6280
6216 REPEAT
6220 PROCgoto_man(Q%)
6225 PROCfind_shafts(A%(Q%,2))
6230 IF ABS(X2%-A%(Q%,1))<ABS(X1%-A%(Q%,1))
     AND X2%<>0 THEN PROCdrag(X2%) ELSE
     PROCdrag(X1%)
6235 PRINT TAB(A%(Q%,1)+2,A%(Q%,2));SPC(1);
6236 SOUND 0,-15,6,3
6240 PROCdown_shaft
6245 PRINT TAB(XM%,YM%);CHR$(228);
6250 Q%=Q%+1
6260 UNTIL Q%=4
6270 PROCgoto_man(Q%)
6280 PROCdrag(5)
6290 ENDPROC
```

PROCgoto_man(Q%) moves ant Q% to the current position of the man. Once the ant has reached the man, line 6225 calls PROCfind_shafts to locate the position of the shafts. If there is only one shaft then its position is returned in X1% and X2% is set to zero. If there are two shafts connecting a tunnel to the one below their positions are returned in X1% and X2%. Once the shafts have been located line 6230 tests to discover which is closer to the current position of the ant that has the man. Then PROCdrag is called to move the ant plus the man to the

closest shaft. Finally PROCdown_shaft is called to drop the man down the shaft. The whole sequence is repeated until the man reaches the bottom level when ant 4 drags the man to the next (line 6280). Quite simple, really!

Of course the simplicity of PROCcapture is due to its use of a number of new procedures. PROCgoto_man merely has to move the ant from its current position to XM%. However, before this move begins the ant has to be pointing in the correct direction.

```
5900 DEF PROCgoto_man(Q%)
5910 IF A%(Q%,1)+1=XM% THEN ENDPROC
5920 IF SGN(XM%-A%(Q%,1)-1)<>A%(Q%,3) THEN
     PROCreverse(Q%)
5930 REPEAT
5940 PRINT TAB(A%(Q%,1),A%(Q%,2));SPC(2);
5950 A%(Q%,1)=A%(Q%,1)+A%(Q%,3)
5960 PRINT TAB(A%(Q%,1),A%(Q%,2));A$(Q%);
5965 PROCdelay
5970 UNTIL A%(Q%,1)+1=XM%
5980 ENDPROC
```

Line 3920 checks to see if the ant is already facing toward the man; if not, PROCreverse is called. After this the REPEAT loop (lines 5930 to 5970) moves the ant one location at a time until it is in contact with the man. PROCfind_shafts works by examining the row of character locations just below the ant's tunnel and storing the co-ordinates of any that are black in Xl% and X2%.

```
5600 DEF PROCfind_shafts(Y%)
5610 X%=0
5620 REPEAT
5630 C=FNC(X%,Y%+1)
5640 X1%=X%
5650 X%=X%+1
5660 UNTIL X%>39 OR C=0
5670 X2%=0
5680 REPEAT
5690 C=FNC(X%,Y%+1)
5700 IF C=0 THEN X2%=X%
5710 X%=X%+1
5720 UNTIL X%>39 OR C=0
5730 ENDPROC
```

The first REPEAT loop (lines 5620 to 5660) searches for the first shaft and the second REPEAT loop (lines 5680 to 5720) searches for the second shaft. If there is no second shaft then the second loop finishes when the edge of the screen is reached and returns with X2% set to

zero.

The third new procedure used by PROCcapture is PROCdrag which is very similar to PROCgoto_man.

```
5800 DEF PROCdrag(X%)
5805 IF A%(Q%,1)+2=X% THEN XM%=X%:ENDPROC
5810 IF SGN(X%-A%(Q%,1)-2)<>A%(Q%,3) THEN
     PROCreverse(Q%)
5820 REPEAT
5830 PRINT TAB(A%(Q%,1),A%(Q%,2));SPC(3);
5840 A%(Q%,1)=A%(Q%,1)+A%(Q%,3)
5850 PRINT TAB(A%(Q%,1),A%(Q%,2));A$(Q%);
     CHR$(228);
5855 PROCdelay
5860 UNTIL A%(Q%,1)+2=X%
5870 XM%=X%
5880 ENDPROC
```

The only real difference between PROCdrag and PROCgoto_man is that PROCdrag moves the ant and the man to the nearest shaft whereas goto-man moves only the ant to the position of the man. To allow the player enough time to see what is happening PROCgoto_man and PROCdrag both use PROCdelay to slow things down a little.

```
9300 DEF PROCdelay
9310 LOCAL T
9320 FOR T=0 TO 100
9330 NEXT T
9340 ENDPROC
```

Making the game better

After playing a few games of Ant Hill it quickly becomes obvious that it it is very easy to win! A game that is too easy is no game at all and at this point many programmers might be tempted to give Ant Hill up as a failure. In fact the first version of most games fails to perform as well as expected for one reason or another, and the next step in developing any game is to work out what is wrong and put it right.

The first impulse is to try to make the game more difficult by decreasing the time allowed. If you try this you will find that progressively decreasing the time allowed has very little effect until you suddenly reach the point where the game is almost impossible. [he trouble is that the game can always be completed in roughly the same amount of time. If you make the time limit greater than this then the

game is easy. If you make the time limit smaller then the game is impossible! To make Ant Hill more interesting to play, we must first find out why it is that a game can be completed in a fixed amount of time.

The most obvious reason is that initially all the ants are moving to the right and as long as the man makes a dash for the left hand tunnels he can get to the next before any of the ants have had much time to turn around! The solution to this is to set the ants moving in random directions at the start of the game. Given that we already have PROCreverse, this is most easily achieved by adding the following line:

```
1215 IF RND(1)>0.5 THEN PROCreverse(I)
```

After this change the game is more difficult and hence more interesting but it is still possible for the man figure to move through any of the ants without being captured and this means that he doesn't really have to worry about exactly where the ants are in the tunnels. The reason for the man being able to move through an ant is that the only time that a capture is tested for is when an ant moves, and for any given ant this only happens, on average, once every five moves of the man. The solution to this is to add a line to check for the man moving onto a position that is already occupied by an ant. This can be done by adding

```
5965 IF FNC(XM%,YM%)<>0 THEN HIT=TRUE
```

to PROCmove_man. This simply tests to make sure that the location that the man is about to move to is black. Once this change is made, another annoying feature becomes apparent. Because an ant can move by up to three character locations at a time, it is possible for an ant to 'jump' over the man. To make this impossible the ants would have to be restricted to a maximum move of two character locations at a time. This can be implemented most simply by changing line 3040 in PROCmove_ant to:

```
3040 A%(Q%,1)=A%(Q%,1)+A%(Q%,3)*2
```

This makes all the ants move by a constant two character locations each time. After making this change the ants still seem to move fast enough and with enough variability to look interesting.

At this point most of the obvious changes have been made and the game is certainly more difficult and more interesting but it still lacks the degree of challenge that a good game should offer. After playing a

number of games it seems that the reason for this is that for much of time the ants are too far away from the shafts to pose any threat to the man. The best games occur when the ants are, by chance, all gathered towards the middle of the screen. This suggests both that the ants' range should be reduced and that the pairs of shafts should be brought closer together. To do this lines 3050 and 3060 in PROCmove_ant have to be changed to:

```
3050 IF A%(Q%,1)<10 THEN A%(Q%,1)=10:
     PROCreverse(Q%)
3060 IF A%(Q%,1)>28 THEN A%(Q%,1)=28:
     PROCreverse(Q%)
```

and line 9010 in PROCshafts changed to:

```
9010 X1=RND(5)+10:X2=RND(5)+25
```

With all of these changes Ant Hill is a game that combines speed with strategy and is good fun to play.

Making the Electron go faster!

Even though Ant Hill is now fun to play it still has one annoying feature. The Electron cannot quite manage to run the program fast enough to make the keypresses that control the man appear to work instantaneously. In fact the keyboard control is sluggish. The solution to this problem is to identify what is taking the most time and convert it from BASIC to assembler.

As a general rule assembler is best avoided and certainly best avoided if the program is at all complicated: complicated things are best done in BASIC! Examining the procedures in PROCone_move suggests that either PROCmove_ant or PROCmove_man should be rewritten in assembler. However, PROCmove_ant is very complicated and so PROCmove_man is the obvious candidate.

On close examination PROCmove_man contains some parts that are complicated and not speed-critical. For example, there is little point in converting PROCdown_shaft to assembler as it needs to run a little slowly! After a little thought, it is clear that the best compromise is to write an assembly language subroutine that replaces the blanking out (line 5030), the testing of the keyboard for the right and left arrow keys and the updating of XM% (lines 3040 and 5050) and the reprinting of

the man character (line 5070). The rest of the action of PROCmove-man, that is, the setting of the colours (lines 5010 and 5020), the testing for the down arrow key and calling PROCdown_shaft (line 5060) and the testing to see if the man has reached the nest (line 5080), are all best implemented in BASIC.

The most obvious place to insert the necessary assembly language and the BASIC to assemble it is at the end of PROCinitialise:

```
1260 DIM CODE% 500
1270 FOR PASS=0 TO 2 STEP 2
1280 P%=CODE%
1285 OSWRCH%=&FFEE
1286 OSBYTE%=&FFF4
1290 [OPT PASS

1320 .MMAN% LDA #31       \TAB(XM%,YM%)
1330 JSR  OSWRCH%
1340 LDA  &70
1350 JSR  OSWRCH%
1360 LDA  &71
1370 JSR  OSWRCH%

1380 LDA  #ASC(" ")       \PRINT " ";
1390 JSR  OSWRCH%

1400 LDA  &70
1410 CMP  #0  \IF XM%=0 THEN GOTO NOLEFT%
1420 BEQ  NOLEFT%

1430 LDA  #129            \INKEY(-26)
1440 LDY  #&FF
1450 LDX  #&E6
1460 JSR  OSBYTE%

1470 CPY  #0  \IF INKEY(-26)=0 THEN
              GOTO NOLEFT%
1480 BEQ  NOLEFT%

1490 DEC  &70 \XM%=XM%-1

1500 .NOLEFT% LDA &70     \IF XM%=39 THEN
                           GOTO NORIGHT%
1510 CMP  #39
1520 BEQ  NORIGHT%

1530 LDA  #129            \INKEY(-122)
1540 LDY  #&FF
1550 LDX  #&86
1560 JSR  OSBYTE%
```

```
1570 CPY  #0  \IF INKEY(-122)=0 THEN
              GOTO NORIGHT%
1580 BEQ  NORIGHT%

1590 INC  &70 \XM%=XM%+1

1600 .NORIGHT%            LDA #31  \TAB(XM%,YM%)
1610 JSR  OSWRCH%
1620 LDA  &70
1630 JSR  OSWRCH%
1640 LDA  &71
1650 JSR  OSWRCH%

1660 LDA  #228            \PRINT CHR$(228);
1670 JSR  OSWRCH%

1680 RTS  \Return to BASIC
1690 ]
1700 NEXT PASS
1800 ENDPROC
```

If you know assembler you should be able to follow this subroutine from the comments written alongside each group of instructions. The comments indicate what the instructions do in terms of their equivalent BASIC commands. The only other information that is important is that the subroutine expects the value of XM% to be stored in memory location &70 and YM% in &71. On return from the subroutine the new value of XM% is stored in &70. (The area &70 to &8F is reserved for use by machine code programs.)

The modifications needed to PROCmove_man are fairly extensive and it is easier to give a listing of the whole procedure rather than just the changes:

```
5000 DEF PROCmove_man
5010 IF YM%>9 THEN COLOUR 128+0 ELSE
     COLOUR 128+1
5020 COLOUR 3
5030 ?&70=XM%:?&71=YM%
5040 CALL MMAN%
5045 XM%=?&70
5060 IF INKEY(-42) AND FNC(XM%,YM%+1)=0 THEN
     PROCdown_shaft
5065 IF FNC(XM%,YM%)<>0 THEN HIT=TRUE
5080 IF XM%=3 AND YM%=26 THEN HOME=TRUE
5090 ENDPROC
```

Line 5030 stores XM% in &70 and YM% in &71 ready for the machine code routine to do its work. Line 5040 calls the routine and line 5045 retrieves the updated value of XM% from &70. Because the machine code routine now prints the man before line 5060 checks for the down arrow key being pressed, it is necessary to add a line to PROCdown_shaft to blank the man again before his descent down the shaft. That is, add

```
5504 PRINT TAB(XM%,YM%);" ";
```

to PROCdown_shaft

After this change to machine code the speed of the program is noticeably faster and the response time of the keyboard is good. A further speed improvement could be achieved by coding part of PROCmove_ant in assembler but this seems unnecessary. At this point Ant Hill is a very pleasing game!

Suggestions for further work

The main part of the Ant Hill program is fairly complete and most of the scope for further work centres on the endgame routines. PROCdestroy nest and PROChatch are both weak when compared to the fascinating performance that the ants go through when PROCcapture is called. The game itself might be improved by making the ants a little more ‘intelligent’ by making them detect and move towards the man when he descends to their level!

The final version – a complete listing

The following listing includes all of the modifications introduced in the text, including the assembly language subroutine. Because of memoy limitations this version will not work with a Electron using disks. If you are using a disk system change to tape by typing *TAPE followed by PAGE=&E00.

```
10 REM ANT HILL
20 MODE 1
30 PROCinitialise
40 PROCtitle
```

```
  50 PROCdraw_scene
  60 REPEAT
  70 PROCone_move
  80 UNTIL TIME_UP OR HIT OR HOME
  90 PROCend_game
 100 CLS
 110 END


1000 DEF PROCinitialise
1010 VDU 23,224,&00,&0E,&0F,&3F,&1F,&33,&44,&84
1020 VDU 23,225,&03,&14,&D8,&FC,&A8,&C0,&40,&10
1030 VDU 23,226,&00,&70,&F8,&FC,&F8,&CC,&22,&21
1040 VDU 23,227,&C0,&28,&1B,&3F,&19,&03,&04,&08
1050 VDU 23,228,&18,&18,&7E,&18,&3C,&66,&C3,&C3
1060 VDU 23,229,&01,&01,&03,&03,&07,&07,&0F,&0F
1070 VDU 23,230,&1F,&1F,&3F,&3F,&7F,&7F,&FF,&FF
1080 VDU 23,231,&80,&80,&C0,&C0,&E0,&E0,&F0,&F0
1090 VDU 23,232,&F8,&F8,&FC,&FC,&FE,&FE,&FF,&FF
1100 VDU 19,0,0,0,0,0:REM 0=BLACK
1110 VDU 19,1,4,0,0,0:REM 1=BLUE
1120 VDU 19,2,3,0,0,0:REM 2=YELLOW
1130 VDU 19,3,1,0,0,0:REM 3=RED
1140 XM%=RND(20)+10
1150 YM%=6
1160 DIM A%(4,3),A$(4)
1170 FOR I=0 TO 4
1180 A%(I,1)=RND(30)+5
1190 A%(I,2)=I*4+10
1200 A%(I,3)=1
1210 A$(I)=CHR$(224)+CHR$(225)
1215 IF RND(1)>0.5 THEN PROCreverse(I)
1220 NEXT I
1230 HIT=FALSE
1240 HOME=FALSE
1250 TIME_UP=FALSE
1260 DIM CODE% 500
1270 FOR PASS=0 TO 2 STEP 2
1280 P%=CODE%
1285 OSWRCH%=&FFEE
1286 OSBYTE%=&FFF4
1290 [OPT PASS
1320 .MMAN%   LDA #31
1330 JSR  OSWRCH%
1340 LDA  &70
1350 JSR  OSWRCH%
1360 LDA  &71
1370 JSR  OSWRCH%
1380 LDA  #ASC(" ")
1390 JSR  OSWRCH%
```

```
1400 LDA  &70
1410 CMP  #0
1420 BEQ  NOLEFT%
1430 LDA  #129
1440 LDY  #&FF
1450 LDX  #&E6
1460 JSR  OSBYTE%
1470 CPY  #0
1480 BEQ  NOLEFT%
1490 DEC  &70
1500 .NOLEFT% LDA        &70
1510 CMP  #39
1520 BEQ  NORIGHT%
1530 LDA  #129 \INKEY(-122)
1540 LDY  #&FF
1550 LDX  #&86
1560 JSR  OSBYTE%
1570 CPY  #0
1580 BEQ  NORIGHT%
1590 INC  &70
1600 .NORIGHT%           LDA #31
1610 JSR  OSWRCH%
1620 LDA  &70
1630 JSR  OSWRCH%
1640 LDA  &71
1650 JSR  OSWRCH%
1660 LDA  #228
1670 JSR  OSWRCH%
1680 RTS
1690 ]
1700 NEXT PASS
1800 ENDPROC

2000 DEF PROCdraw_scene
2010 COLOUR 128+3
2020 CLS
2030 COLOUR 128+0
2040 FOR X=0TO 39
2050 PRINT TAB(X,10);" ";
2060 PRINT TAB(X,14);" ";
2070 PRINT TAB(X,18);" ";
2080 PRINT TAB(X,22);" ";
2090 PRINT TAB(X,26);" ";
2100 NEXT X
2120 COLOUR 128+1
2130 FOR X=0 TO 39
2140 FOR Y=0 TO 6
2150 PRINT TAB(X,Y);" ";
2160 NEXT Y
2170 NEXT X
2175 COLOUR 128+0
```

```
2180 PROCshafts(2,7,9)
2190 PROCshafts(1,11,13)
2200 PROCshafts(2,15,17)
2220 PROCshafts(1,19,21)
2230 PROCshafts(2,23,25)
2240 COLOUR 3:COLOUR 128+1
2250 PRINT TAB(XM%,YM%);CHR$(228);
2260 COLOUR 2:COLOUR 128+0
2270 PRINT TAB(1,25);CHR$(229);CHR$(231);
2280 PRINT TAB(1,26);CHR$(230);CHR$(232);
2290 COLOUR 2
2300 FOR I=0 TO 4
2310 PRINT TAB(A%(I,1),A%(I,2));A$(I);
2320 NEXT I
2330 TIME=0
2340 ENDPROC

3000 DEF PROCmove_ant
3005 COLOUR 128+0:COLOUR 2
3010 Q%=RND(5)-1
3020 PRINT TAB(A%(Q%,1),A%(Q%,2));SPC(2);
3030 IF RND(1)<.01 THEN PROCreverse(Q%)
3040 A%(Q%,1)=A%(Q%,1)+A%(Q%,3)*2
3050 IF A%(Q%,1)<10 THEN A%(Q%,1)=10:
     PROCreverse(Q%)
3060 IF A%(Q%,1)>28 THEN A%(Q%,1)=28:
     PROCreverse(Q%)
3070 PRINT TAB(A%(Q%,1),A%(Q%,2));A$(Q%);
3080 HIT=FNgot_man(Q%)
3090 ENDPROC

3500 DEF PROCreverse(Q%)
3510 IF A$(Q%)=CHR$(224)+CHR$(225) THEN
     A$(Q%)=CHR$(227)+CHR$(226)
     ELSE A$(Q%)=CHR$(224)+CHR$(225)
3520 A%(Q%,3)=-A%(Q%,3)
3530 ENDPROC

4000 DEF PROCone_move
4010 PROCmove_man
4020 PROCmove_ant
4030 PROCtime
4040 ENDPROC



4500 DEF PROCtime
4510 PRINT TAB(2,31);TIME/100;
4520 IF TIME>MAX THEN TIME_UP=TRUE
4530 ENDPROC
```

```
5000 DEF PROCmove_man
5010 IF YM%>9 THEN COLOUR 128+0 ELSE
     COLOUR 128+1
5020 COLOUR 3
5030 ?&70=XM%:?&71=YM%
5040 CALL MMAN%
5045 XM%=?&70
5060 IF INKEY(-42) AND FNC(XM%,YM%+1)=0 THEN
     PROCdown_shaft
5065 IF FNC(XM%,YM%)<>0 THEN HIT=TRUE
5080 IF XM%=3 AND YM%=26 THEN HOME=TRUE
5090 ENDPROC

5500 DEF PROCdown_shaft
5504 PRINT TAB(XM%,YM%);" ";
5505 COLOUR 128+0
5510 FOR I=1 TO 4
5520 YM%=YM%+1
5530 PRINT TAB(XM%,YM%);CHR$(226);
5540 SOUND 1,-15,148-I%*4,2
5550 IF ADVAL(-6)<>15 THEN GOTO 5550
5560 PRINT TAB(XM%,YM%);" ";
5570 NEXT I
5580 ENDPROC

5600 DEF PROCfind_shafts(Y%)
5610 X%=0
5620 REPEAT
5630 C=FNC(X%,Y%+1)
5640 X1%=X%
5650 X%=X%+1
5660 UNTIL X%>39 OR C=0
5670 X2%=0
5680 REPEAT
5690 C=FNC(X%,Y%+1)
5700 IF C=0 THEN X2%=X%
5710 X%=X%+1
5720 UNTIL X%>39 OR C=0
5730 ENDPROC

5800 DEF PROCdrag(X%)
5805 IF A%(Q%,1)+2=X% THEN XM%=X%:ENDPROC
5810 IF SGN(X%-A%(Q%,1)-2)<>A%(Q%,3) THEN
     PROCreverse(Q%)
5820 REPEAT
5830 PRINT TAB(A%(Q%,1),A%(Q%,2));SPC(3);
5840 A%(Q%,1)=A%(Q%,1)+A%(Q%,3)
5850 PRINT TAB(A%(Q%,1),A%(Q%,2));A$(Q%);
     CHR$(228);
5855 PROCdelay
```

```
5860 UNTIL A%(Q%,1)+2=X%
5870 XM%=X%
5880 ENDPROC

5900 DEF PROCgoto_man(Q%)
5910 IF A%(Q%,1)+1=XM% THEN ENDPROC
5920 IF SGN(XM%-A%(Q%,1)-1)<>A%(Q%,3) THEN
     PROCreverse(Q%)
5930 REPEAT
5940 PRINT TAB(A%(Q%,1),A%(Q%,2));SPC(2);
5950 A%(Q%,1)=A%(Q%,1)+A%(Q%,3)
5960 PRINT TAB(A%(Q%,1),A%(Q%,2));A$(Q%);
5965 PROCdelay
5970 UNTIL A%(Q%,1)+1=XM%
5980 ENDPROC

6000 DEF PROCend_game
6010 IF HIT THEN PROCcapture:GOTO 6100
6020 IF TIME_UP THEN PROChatch:GOTO 6100
6030 IF HOME THEN PROCdestroy_nest
6100 VDU 23,1,1;0;0;0;
6110 PRINT TAB(2,29);"Another game ";:INPUT A$
6120 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 6100
6130 IF LEFT$(A$,1)="Y" THEN RUN
6140 ENDPROC


6200 DEF PROCcapture
6210 PRINT TAB(2,30);"You are captured!"
6215 IF Q%=4 THEN GOTO 6280
6216 REPEAT
6220 PROCgoto_man(Q%)
6225 PROCfind_shafts(A%(Q%,2))
6230 IF ABS(X2%-A%(Q%,1))<ABS(X1%-A%(Q%,1))
     AND X2%<>0 THEN PROCdrag(X2%) ELSE
     PROCdrag(X1%)
6235 PRINT TAB(A%(Q%,1)+2,A%(Q%,2));SPC(1);
6236 SOUND 0,-15,6,3
6240 PROCdown_shaft
6245 PRINT TAB(XM%,YM%);CHR$(228);
6250 Q%=Q%+1
6260 UNTIL Q%=4
6270 PROCgoto_man(Q%)
6280 PROCdrag(5)
6290 ENDPROC

6500 DEF PROChatch
6510 PRINT TAB(2,30);"THE ANTS ARE HATCHING "
6520 FOR I=1 TO RND(20)+20
6530 X=RND(25)+5:Y=RND(10)+10
```

```
6540 PRINT TAB(X,Y);A$(RND(5)-1);
6550 SOUND 0,-15,RND(4)+3,RND(2)
6560 NEXT I
6570 ENDPROC

6800 DEF PROCdestroy_nest
6810 PRINT TAB(2,30);"YOU DID IT!"
6820 DATA 72,.5,80,.5,88,.5,96,.5,100,1,100,1
6840 ENDPROC
6890 DATA 999,999
6900 READ P,D
6910 IF P=999 THEN GOTO 6940
6920 SOUND 1,-15,P,D*10
6925 SOUHD 1,0,0,1
6930 GOTO 6900
6940 ENDPROC


8000 DEF PROCtitle
8010 CLS:COLOUR 128+0:COLOUR 2
8020 PRINT TAB(12,3);"A N T   H I L L"
8030 PRINT STRING$(20,A$(1))
8040 PRINT TAB(5,10);"In this game you have a
     fixed"
8050 PRINT "amount of time to destroy the ants
     nest"
8060 PRINT "before the eggs hatch!"
8070 PRINT
8080 PRINT "Use the arrow keys to move but"
8090 PRINT "BEWARE: the soldier ants guarding
     the"
8095 PRINT "tunnels will capture you and take
     you"
8096 PRINT "to the nest to feed the young!"
8100 PRINT
8110 PRINT "Good luck"
8300 PRINT TAB(5,25);"Which difficulty level -"
8305 PRINT
8310 PRINT "1. Expert, 2. Medium or 3. Novice";
8320 INPUT DF
8330 IF DF<1 OR DF>3 THEN GOTO 8300
8340 MAX=(20+DF*10)*100
8350 VDU 23,1,0;0;0;0;
8360 ENDPROC


9000 DEF PROCshafts(NO,Y1,Y2)
9010 X1=RND(10)+4:X2=RND(10)+25
9020 FOR Y=Y1 TO Y2
9030 IF NO=1 THEN PRINT TAB((X1+X2)/2,Y);" ";
9040 IF NO=2 THEN PRINT TAB(X1,Y);" ";
```

```
     TAB(X2,Y);" ";
9050 NEXT Y
9060 ENDPROC

9100 DEF FNC(X%,Y%)
9110 X%=3+32*X%
9120 Y%=1020-32*Y%
9130 =POINT(X%,Y%)

9200 DEF FNgot_man(Q%)
9210 IF A%(Q%,2)<>YM% THEN =FALSE
9220 IF A%(Q%,1)<>XM% AND A%(Q%,1)+1<>XM%
     THEN =FALSE
9230 =TRUE

9300 DEF PROCdelay
9310 LOCAL T
9320 FOR T=0 TO 100
9330 NEXT T
9340 ENDPROC
```

Chapter Three
Leap Frog

Leap Frog is a game that involves full two-dimensional animation. That is, one of the objects in the game, the frog, moves both horizontally and vertically at the same time. (This should be compared to Ant Hill in the last chapter where the objects only moved either horizontally or vertically.) In addition, a number of other interesting animation techniques are used to create a fascinating display using surprisingly few BASIC statements. The game itself is both fun to watch and play and provides a host of possibilities for variations.

The game design

Leap Frog uses many of the elements to be found in one of the first computer games, variously called 'Break Out', 'Little Brick Out', 'Knock 'Em Down' and many more names. These games are played by bouncing a ball around the screen by means of a bat with the objective of erasing as many of the coloured blocks printed in a band at the top of the screen as possible. In the simplest versions of the game the skill is simply in moving the bat to intercept the ball and so keep it bouncing. In the more sophisticated versions the player can direct the ball by hitting it with different parts of the bat.

In Leap Frog the coloured blocks or bricks are replaced by rows of insect shapes and the ball is replaced by a jumping frog. Obviously it would be cruel to use a simple bat to keep the frog moving and so a seesaw is used instead. A second frog sits on the seesaw and when the first frog lands, this second frog is catapulted into the air and the first frog sits on the seesaw, awaiting its turn.

Another difference between the traditional game of Breakout and Leap Frog is to be found in the path that the frog takes as it flies through the air. In Breakout the ball moves in straight lines, bouncing its way around the screen, but in Leap Frog the frog moves as if it were being pulled back to the ground by gum fly. Each time the frog lands on the seesaw the frog is catapulted into the air bounces a little higher, so

to reach the back row of insects it is necessary to keep the frogs bouncing. This is all there is to the basic Leap Frog game but before going on to implement it in BASIC it is worth looking at the principles of making a sprite bounce around the screen and move under simulated gravity. Figure 3.1 gives an impression of the screen display when the program runs.

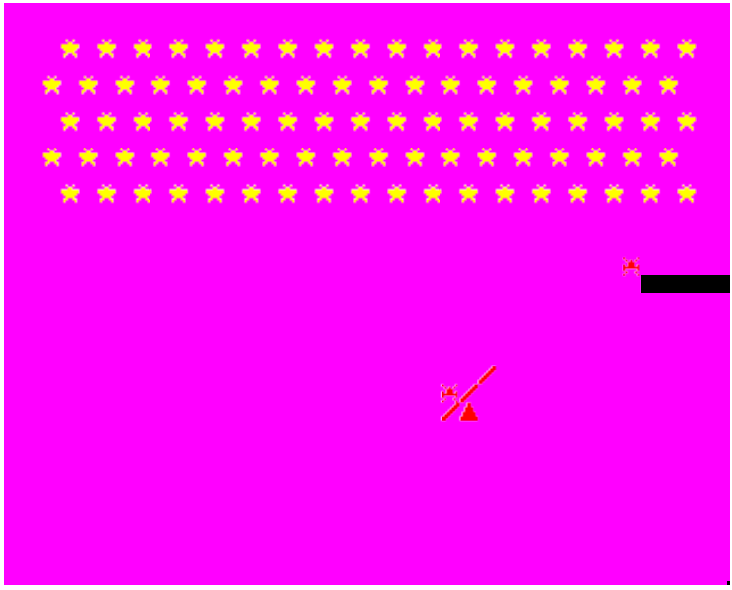


Fig. 3.1.

Bouncing and falling

The idea of animating an object by repeatedly updating its co-ordinates by adding quantities that correspond to horizontal and vertical velocities has already been described in Chapter Two. That is, the basic animation cycle is:

```
PRINT TAB(X,Y);" ";:REM erase object
X=X+VX:Y=Y+VY      :REM update co-ordinates
PRINT TAB(X,Y);C$; :REM reprint object
```

where X and Y hold the object's position and VX and VY hold the object's horizontal and vertical velocities. Using this scheme the object will appear to move across the screen following the same line of motion until it disappears off the screw. This sort of motion is not really very useful in a game and it is not long before the need to bounce an object arises. In principle bouncing an object off a vertical or horizontal boundary is easy. If a sprite moving with velocity VX,VY strikes a horizontal boundary then its velocity changes to VX,VY. That is, a bounce off a horizontal boundary reverses the vertical velocity. Similarly, a bounce off a vertical boundary reverses the horizontal velocity. As an example of this principle try the following program:

```
10 MODE 1
20 VX=1:VY=1
30 X=RND(38):Y=RND(29)
40 PRINT TAB(X,Y);" ";
50 X=X+VX:Y=Y+VY
60 IF X<1 OR X>38 THEN VX=-VX
70 IF Y<1 OR Y>29 THEN VY=-VY
80 PRINT TAB(X,Y);"A":
90 GOTO 40
```

Line 60 detects collisions with the vertical edges of the screen and reverses the horizontal velocity. Line 70 detects collisions with the horizontal edges of the screen and reverses the vertical velocity. The overall result is a letter 'A' that bounces around the screen! What is surprising is that this sort of animation takes so few lines of BASIC.

Nearly all other types of complicated sprite motion can be produced by changing the horizontal and vertical velocities each time through the animation loop. For example, to make a sprite move across the screen as if it were under the influence of gravity all we have to do is change the vertical velocity by a fixed amount each time through the loop. This mimics what happens when, for example, a real ball is thrown in the air. The ball starts out with a certain vertical velocity that is constantly reduced by gravity until it reaches zero and then changes direction, so bringing the ball back to earth faster and faster. Notice that gravity doesn't affect the ball's horizontal velocity at all. The only factors that change the ball's horizontal velocity are air resistance or wind and in most cases these can be ignored. The following program makes a sprite move like a thrown ball:

```
10 MODE 1
20 X=0:Y=31
30 VX=.1:VY=-.1
```

```
40 A=.0005
50 PRINT TAB(X,Y);" ";
60 X=X+VX:Y=Y+VY
70 VY=VY+A
80 PRINT TAB(X,Y);"A":
90 GOTO 50
```

Line 60 performs the usual co-ordinate update but in this program line 70 also updates the vertical velocity by adding a constant to it. The result is a letter 'A' that follows a parabolic path and then goes off the bottom of the screen. The size of the constant used in line 40 controls the force of 'gravity' that the letter A responds to. Increasing it will bring the A back down more quickly. Notice that to allow for the effect of gravity it is necessary to use co-ordinates that are not integers and this can cause complications.

The main program

After this discussion of the theoretical issues involved it is time to return to the details of the Leap Frog program. Once again, mode 1 seems to be the best choice, for the colour and resolution it offers. The elements of the game are straightforward. The player is given ten frogs with which to attempt to eat all the insects printed at the top of the screen. Each new frog starts off by jumping of a platform and the user has to position the seesaw to catapult the second frog into the air. Once started in this way the main animation loop continues until the user 'misses' a frog with the seesaw. A new frog then appears and the game continues. There are two ways in which the game can end, either by the player managing to eat all the insects or by using up all ten frogs. The main program is:

```
  10 REM Leap Frog
  20 MODE 1
  30 PROCinit
  40 PROCtitle
  50 PROCprint_flies
  60 PROCstartup
  80 REPEAT
  90 PROCmove_ss
 100 PROCmove_f
 110 UNTIL GAME_END
 120 PROCend_game
 130 IF AGAIN THEN RUN
 140 END
```

PROCinit sets up the user-defined characters, logical and physical colour assigments and other constants used later in the program. PROCtitle prints a screen of instructions and asks the user to choose a difficulty level. PROCprint flies prints five rows of insects at the top of the screen ready for the game to commence. PROCstartup makes a frog jump off the platform while the player attempts to position the seesaw underneath it. The main animation loop is formed by lines 80 to 110, PROCmove_ss allows the player to move the seesaw using the left and right arrow keys and PROCmove_f moves the frog, taking account of gravity and any bouncing that is necessary. Finally, PROCend_game sums up the player's performance and asks if another game is required.

PROCinit and PROCtitle

The middle section of PROCinit looks a little complicated but in fact the procedure is not at ail difficult to understand:

```
1000 DEF PROCinit
1010 VDU 23,224,&03,&07,&0E,&1C,&38,&70,&E0,&C0
1020 VDU 23,225,&C0,&E0,&70,&38,&1C,&0E,&07,&03
1030 VDU 23,226,&18,&18,&3C,&3C,&7E,&7E,&FF,&FF
1040 VDU 23,227,&24,&18,&FF,&FF,&7E,&3C,&66,&42
1050 VDU 23,228,&82,&54,&38,&BA,&FE,&82,&44,&82
1060 VDU 19,0,5,0,0,0,0:REM 0=MAGENTA
1070 VDU 19,1,5,0,0,0,0:REM 1=YELLOW
1080 VDU 19,2,1,0,0,0,0:REM 2=RED
1090 VDU 19,3,0,0,0,0,0:REM 3=BLACK
1100 L$="   "+CHR$(224)+" "+CHR$(10)+
     STRING$(5,CHR$(8))
1110 L$=L$+"  "+CHR$(224)+"  "+CHR$(10)+
     STRING$(6,CHR$(8))
1120 L$=L$+"  "+CHR$(224)+CHR$(226)+"  "
1130 R$=" "+CHR$(225)+"   "+CHR$(10)+
     STRING$(5,CHR$(8))
1140 R$=R$+"  "+CHR$(225)+CHR$(10)+
     STRING$(3,CHR$(8))
1150 R$=R$+"  "+CHR$(226)+CHR$(225)+" "
1160 L$=CHR$(17)+CHR$(2)+R$
1170 R$=CHR$(17)+CHR$(2)+R$
1200 M%=1
1210 S$=L$
1220 X%=23
1260 FROG=1
1270 GAME_END=FALSE
1280 SC=0
1290 ENDPROC
```

Lines 1010 to 1050 define the five shapes used by Leap Frog, which are illustrated in Fig. 3.2. CHR$(224) and CHR$(226) to go together to form the left seesaw and CHR$(225) and CHR$(226) go together to form the right seesaw. CHR$(227) is the insect shape and CHR$8(229) is the frog shape.

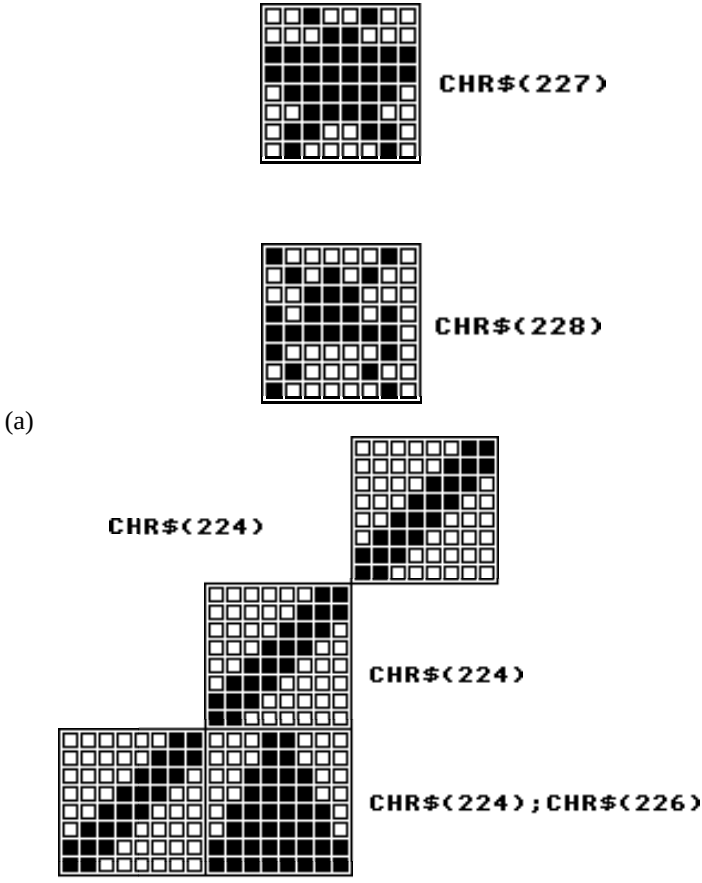


Fig. 3.2. Graphics characters for (a) insect and frog. (b) Right Seesaw

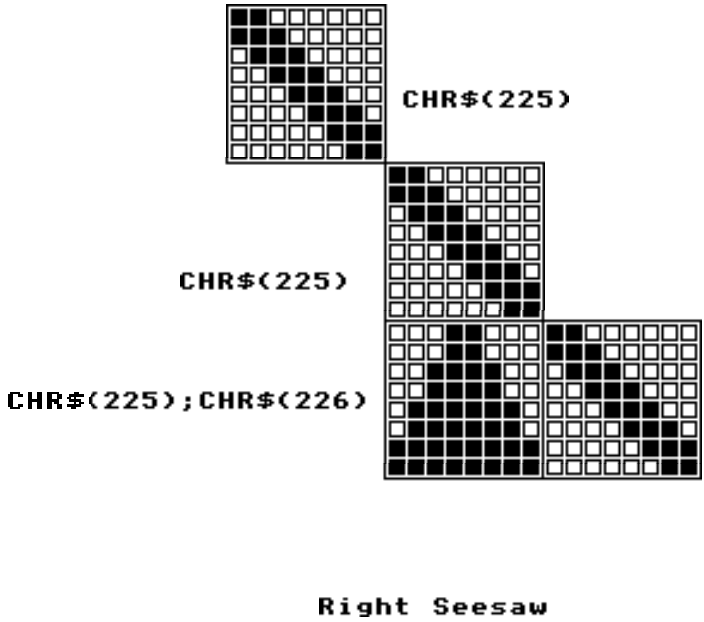


Fig. 3.2. (cont.). Graphics characters for (c) Left Seesaw

Lines 1060 to 1090 set up the physical to logical colour assignment. Magenta is used for the background, yellow for the insects, red for the seesaw and black for the frogs.

Lines 1100 to 1170 define two arrays, L$ and R$, which contain a sequence of characters that print the left and right seesaws respectively. The technique used here is worth studying because it is a simple solution to many graphics problems that involve printing a number of characters to form a larger shape. If you examine line 1100 you will se that it defines the top line of the left seesaw shown in Fig. 3.2. The spaces are included so that the seesaw is 'self-blanking' as it moves from side to side. The CHR$(10) is a control code that moves the cursor down one line and STRING$(5,CHR$(8)) appends 5 cursor left control codes to the string. If you think about it you should be able to see that this sequence of control codes results in the cursor being positioned one line down and below the first character (a blank) printed from the string. Line 1110 stores the characters that make up the next row of the seesaw character in L$, along with the necessary control codes to move the cursor down to the next line and under the first character printed. Finally, line 1120 stores the last line of the seesaw in L$. In the same way the characters and cursor control codes that form

the right seesaw are stored in R$ by lines 1130 to 1150.

To complete our explanation of the use of VDU control codes, lines 1160 and 1170 store a pair of codes that automatically set the foreground colour to red. CHR$(17) has the same effect as a COLOUR statement and the code which follows it is taken by the VDU driver to be a colour code. Thus CHR$(l7)+CHR$(2) has the same effect as COLOUR 2. Embedding control codes in a string of printable characters is a a very powerful graphics technique and should always be kept in mind when writing programs.

Lines 1300 to 1280 define the initial values of a number of variables used in the rest of the program. M% is used as an indicator of which seesaw, left or right, is currently in use. If M%, is 1 then the left seesaw is is on the screen and if M% is 0 the right seesaw is on the screen. S$ is the string that hold the characters that produce the current seesaw shape. That is, PRINT TAB(X,Y);S$; prints the current seesaw shape with its top left hand corner at X,Y. X% gives the horizontal position of the seesaw and it is set to 23 by line 1220 to give the initial position of the seesaw when the game starts. GAME_END is used to signal the end of the game and SC is used to hold the current score.

PROCtitle simply prints some instructions and asks for the difficulty level in D to be input:

```
6000 DEF PROCtitle
6010 COLOUR 128+0
6020 CLS
6030 COLOUR 3
6040 PRINT TAB(10,5);"L E A P   F R O G"
6050 PRINT TAB(0,10);
6060 PRINT " In this game you must eat the
     flies ";CHR$(227)
6065 PRINT
6070 PRINT " by jumping your two frogs ";
     CHR$(228);" off the"
6075 PRINT
6080 PRINT " seesaw using the left and right
     arrow"
6085 PRINT
6090 PRINT " keys - you have ten frogs to
     clear"
6095 PRINT
6100 PRINT " the screen ...."
6110 PRINT TAB(5,25);"Difficulty level "
6120 PRINT TAB(8);"1 (difficult) to 10
     (easy)";
6125 INPUT D
6130 IF D<1 OR D>10 THEN GOTO 6020
6140 ENDPROC
```

PROCprint_flies

PROCprint_flies is a simple procedure that prints five rows of insects:

```
2000 DEF PROCprint_flies
2010 COLOUR 128+0:COLOUR 1
2020 CLS
2030 C%=0
2040 FOR V=2 TO 10 STEP 2
2050 C%=NOT C%
2060 FOR U=2 TO 36 STEP 2
2070 PRINT TAB(U-C%,V);CHR$(227);
2080 NEXT U
2090 NEXT V
2100 ENDPROC
```

An interesting technique is the use of C% in line 2070 to determine the spacing of alternate rows. To understand how this works all you need to know is that if C% is 0 then NOT C% is –1.

PROCstartup

After PROCprint_flies, PROCstartup prints a frog on a ledge at the right hand side of the screen and steadily moves it to the edge. When the frog reaches the edge it jumps downward. If the seesaw is under the falling frog then the game continues; if not, then PROCstartup is called again from within the animation loop.

```
5000 DEF PROCstartup
5010 PROCmove_ss
5015 VDU 23,1,0;0;0;0;
5020 COLOUR 128+3
5030 PRINT TAB(35,15)STRING$(5," ");
5040 FOR F=39 TO 34 STEP -.15
5050 PRINT TAB(F,14);CHR$(228);" ";
5060 PROCmove_ss
5070 NEXT F
5080 FOR G=14 TO 20 STEP .5
5090 PRINT TAB(34,G-1);" ";
5100 PRINT TAB(34,G);CHR$(228)
5110 PROCmove_ss
5120 NEXT G
5130 F=34
5140 G=20
5150 A=.4+RND(0)/10
```

```
5160 B=-(.1+RND(0)/2.5)
5165 P=0.008
5170 PRINT TAB(2,30);"FROG ";FROG;" SCORE=";SC
5180 A=A+A/D
5190 B=B+B/D
5200 P=P+P/D
5210 PROCseesaw
5220 ENDPROC
```

Line 5010 calls PROCmove_ss which in this case simply serves to draw the seesaw at its initial position. Then line 5030 draws a short black line at the right hand side of the screen to act as the ledge for the frog to jump off. The first FOR loop, lines 5040 to 3070, moves the frog along the ledge. Each time through the loop, PROCmove_ss is called, line 5060, to allow the player an opportunity to move the seesaw. The second FOR loop, lines 5080 to 5120, moves the frog vertically downward. Finally, lines 5130 to 5200 initialise variables used by the the rest of the program. F and G are used to record the current position of the frog. That is, the frog at TAB(F,G). The variables A and B are the vertical and horizontal velocity of the frog. Both are set to random values – lines 5150 and 5160 respectively. Line 5165 sets P, the 'gravity constant' to 0.008. Lines 5180 to 5200 adjust the velocities and gravity constant to take account of the difficulty level chosen by the player. Essentially, making D larger increases the horizontal and vertical velocity and so makes the frog move faster. Finally, PROCstartup calls PROCseesaw to test if the falling frog has landed on the seesaw or not.

PROCmove_ss and PROCmove_f

PROCmove_ss allows the player to move the seesaw using the arrow keys. It is surprisingly simple, mainly because of the use of strings containing control codes to print the seesaw (see PROCinit for more details).

```
3000 DEF PROCmove_ss
3010 IF INKEY(-26) AND X%>0 THEN X%=X%-1
3020 IF INKEY(-122) AND X%<35 THEN X%=X%+1
3030 PRINT TAB(X%,20);S$;
3040 IF M%=0 THEN K%=X%+3:F$=CHR$(228)+" "
     ELSE K%=X%:F$=" "+CHR$(228)
3050 COLOUR 3
```

```
3060 PRINT TAB(K%,21);F$;
3070 ENDPROC
```

Lines 3010 and 3020 test for the right and lelt arrow keys and update the horizontal position of the seesaw (stored in X%,). Line 3030 prints the seesaw at its new position. Notice that there is no need explicitly to blank out the old seesaw because of the blanks included at each end of the seesaw. Lines 3040 to 3060 print the second frog on the low side of the seesaw ready to be catapulted into the air. Line 3040 sets the horizontal position of this frog, taking into account which version (right or left) of the seesaw is being used. Notice that there is no need to set the foreground colour before printing the seesaw because the necessary control codes for a red foreground are included in the string. However, the foreground colour for the frog has to be set using COLOUR (line 3050).

PROCmove_f is fairly straightforward and follows the theory for bouncing and falling objects at the beginning of this chapter.

```
4000 DEF PROCmove_f
4010 PRINT TAB(F,G);" ";
4020 F=F+B:G=G+A:A=A+P
4030 IF F>38 OR F<2 THEN B=-B
4040 IF G<2 THEN G=2:A=-A
4050 IF G<11 THEN PROChit_fly
4060 PRINT TAB(F,G);CHR$(228);
4070 IF G>20 THEN PROCseesaw
4080 ENDPROC
```

Line 4010 blanks out the frog at the old position. Line 4020 performs the position and velocity update. (As you will recall, the frog's horizontal position is stored in F, its horizontal velocity in B, its vertical position in G, its vertical velocity in A and the gravity constant is stored in P.) Line 4040 deals with bounces off the top edge of the screen. Line 4050 checks to see if the frog is high enough up the screen to have hit a fly. Line 4070 checks to see if the frog is low enough on the screen to have landed on the seesaw and calls PROCseesaw if this is so. The reprinting of the frog is taken care of by line 4060 and line 4070 checks to see if the frog is low enough down the screen to have landed on the seesaw. If so, PROCseesaw is called.

PROCseesaw, PROChit_fly and PROCmiss

PROCseesaw tests to see if the frog has landed on the seesaw or not:

```
4500 DEF PROCseesaw
4510 D%=F-X%-M%
4520 IF D%<1 OR D%>2 THEN PROCmiss:ENDPROC
4530 IF M%=1 THEN S$=R$:M%=0 ELSE S$=L$:M%=1
4540 A=-A
4550 G=20:F=K%+1
4560 SOUND 1,-15,200,1
4570 ENDPROC
```

As the horizontal position of the frog is in F and the horizontal position of the top left hand corner of the seesaw is in X% the difference between the two can be used to discover if the frog has landed on the seesaw or not. The only problem is that the side of the seesaw that the frog has to land on depends on whether it is a left hand or right hand seesaw. If the seesaw is a left hand one then the difference has to be 2 or 3 for a correct landing. If the seesaw is a right hand one then the difference has to be 1 or 2 for a correct landing. By subtracting the variable M%, which is 1 for a left hand seesaw and 0 otherwise, this difference can be made to be only either 1 or 2 for a correct landing. This 'corrected' difference is calculated by line 4510 and the test for a for a correct landing is carried out by line 4520 which calls PROCmiss if the frog has not landed on the seesaw. If the frog has landed successfully on the seesaw line 4530 changes the left seesaw into the right seesaw and vice versa. Line 4540 reverses the frog's vertical velocity, effectively bouncing it off the seesaw. Line 4550 updates the frog's co-ordinates to the position of the frog on the other end of the seesaw.

If the frog fails to land on the seesaw PROCmiss is called to bring a new frog into play unless, of course, all ten frogs have already been used.

```
8000 DEF PROCmiss
8005 PRINT TAB(F,G);" ";
8010 FROG=FROG+1
8020 IF FROG>10 THEN GAME_END=TRUE:ENDPROC
8025 SOUND 1,-15,50,5
8030 IF M%=0 THEN M%=1:S$=L$
8040 PROCstartup
8050 ENDPROC
```

If ten frogs have been used line 8020 sets GAME_END to TRUE and thus brings the game to an end. Otherwise the seesaw is set to a left hand version if necessary by line 8030 and then line 8040 calls PROCstartup to continue the game with a new frog.

PROChit_fly simply uses the FNC function (described in Chapter Two when it was used in the Ant Hill program) to test the colour of a

point at the centre of the character loaction that the frog is about to move onto. If it is yellow then the frog has hit a fly.

```
7000 DEF PROChit_fly
7010 IF FNC(F,G)<>1 THEN ENDPROC
7020 SOUND 1,-15,100,1
7030 SC=SC+1
7040 PRINT TAB(2,30);"FROG ";FROG;" SCORE=";SC
7050 IF SC=76 THEN GAME_END=TRUE
7060 ENDPROC
```

If the frog has hit a fly then all that happens is that the score is update (line 7030) and checked to see if all the flies have been eaten (line 7050) The function FNC is defined at the very end of the program:

```
9500 DEF FNC(X%,Y%)
9510 X%=16+32*X%
9520 Y%=1011-32*Y%
9530 =POINT(X%,Y%)
```

PROCend_game

PROCend_game simply prints a number of messages depending on the final score and then asks if another game is required.

```
9000 DEF PROCend_game
9005 CLS
9010 COLOUR 3
9020 PRINT TAB(1,10);
9030 IF SC<10 THEN PRINT "A mere tadpole could
     do better":GOTO 9400
9040 IF SC<20 THEN PRINT "Were your frogs
     three-legged?":GOTO 9400
9050 IF SC<40 THEN PRINT "Not bad for a
     shortsighted toad":GOTO 9400
9060 IF SC<50 THEN PRINT "Pretty good
     considering....":GOTO 9400
9070 IF SC<60 THEN PRINT "A great fly eating
     operation"
9080 PRINT "Try entering the frog olympics"
9400 PRINT TAB(5,15);"You scored ";SC
9410 PRINT TAB(10,18);
9420 INPUT "Another game (Y/N)",A$
9430 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 9400
9440 IF LEFT$(A$,1)="Y" THEN AGAIN=TRUE ELSE
     AGAIN=FALSE
9450 ENDPROC
```

Evaluation and improvements

The most obvious problem with Leap Frog is that it runs much to slowly! Changing some of the procedures in the animation loop to assembler looks as though it might be the best cure but there are a number of difficulties with this scheme. As the anitmation of the frog involves real variables, PROCmove_f is not at all easy to change to assembler – real arithmetic is something that is usually left to BASIC. Even if PROCmove_f were converted to assembler the chances are that the speed increase would not be very great. PROCmove_ss is a very different matter as the seesaw's co-ordinates are integers and as a result it can be converted into assembler very easily. However if you do write an assembly language subroutine to replace PROCmove_ss, you will discover that the speed increase is still not sufficient to make the game playable! The trouble is that in mode 1 it takes far too many memory accesses to store a single character on the screen. In other words, Leap Frog is trying to write too many characters to a mode 1 screen for the animation loop to run fast enough to be acceptable. The only real solution is to abandon the four-colour mode 1 and use the two-colour mode 4 instead, This is not a difficult change to make to the program. Simply change lines 20, 1070, 1080, and 5020 to read:

```
20 MODE 4

1070 VDU 19,1,0,0,0,0:REM 1=BLACK
1080 COLOUR 128+0:COLOUR 1
```

and

```
5020 COLOUR 128+1
```

It is also necessary to add line 5035:

```
5035 COLOUR 128+0
```

and to delete lines 1090, 1160, 1170, 2010, 3050, 6010, 6030 and 9010. (If you are in any doubt about these changes then look at the final listing at at the end of this chapter which includes all of them.)

Following this change Leap Frog certainly runs fast enough at its most difficult to keep a player busy and slow enough at its easiest for a beginner to learn the game.

After these modifications the game is fun to play, although there is

clearly plenty of room for improvements by way of sound effects and extra animation, and now the main problem with the game is that it only exercises one sort of skill and this can become boring. It is difficult to position the seesaw under the falling frog and this provides the games's initial interest. However, after you have learned how to control seesaw there is very little that you can do to increase the number of insects that you manage to eat other than by keeping the frog bouncing for as long as possible. In some ways the present version of the game wastes the use of a seesaw to catapult the frog into the air because it treats it like a simple bat. However, it is not difficult to change this.

If you think about it for a moment the frog that is catapulted off the seesaw should fly into the air in a direction that depends on which version of the seesaw, left or right, does the catapulting. At the moment the frog leaves the seesaw with the same horizontal velocity as the fro that lands on it. This means that if a frog moving to the left of the scree lands on the seesaw the frog that leaves it continues to move to the left. However it is not difficult to see that a frog that is catapulted off a left hand seesaw should move to the right and one that is catapulted off right hand seesaw should move to the left, no mater which way the incoming frog is moving. It is quite easy to make this change to the game by adding the following line to PROCseesaw:

```
4525 IF M%=l THEN B=ABS(B) ELSE B=-ABS(B)
```

This certainly alters the way that the frog bounces off the seesaw so that it makes the game more interesting but there is still no increase in the skill necessary to play the game. To increase the skill the obvious thing to do is to give the player control over which seesaw is in use. By choosing the seesaw that the frog will land on the player can to a certain extent direct the frog to where there are most insects left. The following additions allow the player to change the seesaw by pressing the arrow key:

```
3025 IF INKEY(-58) THEN PROCflip

4600 DEF PROCflip
4610 IF M%=1 THEN S$=R$:M%=0 ELSE S$=L$:M%=1
4620 TIME=0
4630 REPEAT
4640 UNTIL TIME>10
4650 ENDPROC
```

The time delay in PROCflip (lines 4620 to 4640) is necessary to allow

the up arrow key to change the seesaw at a reasonable rate so that the player can stop pressing the key when the correct seesaw is on the screen.

With this modification the game is altogether different and the play is not only involved in the fairly simple task of moving the seesaw to the correct place but in selecting which seesaw should be used for the frog to eat the flies. There are many more possibilities for using the seesaw to control the way that the frog moves, for example the height of bounce could be made to depend on exactly where on the seesaw the frog landed, the horizontal velocity could also depend on how the seesaw was moving just before landing, the frog could lose velocity each time it eats a fly ... but the choice and implementation of these modifications are left to your imagination.

The final version – a complete listing

```
   10 REM Leap Frog
   20 MODE 4
   30 PROCinit
   40 PROCtitle
   50 PROCprint_flies
   60 PROCstartup
   80 REPEAT
   90 PROCmove_ss
  100 PROCmove_f
  110 UNTIL GAME_END
  120 PROCend_game
  130 IF AGAIN THEN RUN
  140 END

 1000 DEF PROCinit
 1010 VDU 23,224,&03,&07,&0E,&1C,&38,&70,&E0,&C0
 1020 VDU 23,225,&C0,&E0,&70,&38,&1C,&0E,&07,&03
 1030 VDU 23,226,&18,&18,&3C,&3C,&7E,&7E,&FF,&FF
 1040 VDU 23,227,&24,&18,&FF,&FF,&7E,&3C,&66,&42
 1050 VDU 23,228,&82,&54,&38,&BA,&FE,&82,&44,&82
 1060 VDU 19,0,5,0,0,0,0:REM 0=MAGENTA
 1070 VDU 19,1,0,0,0,0,0:REM 1=BLACK
 1080 COLOUR 128+0:COLOUR 1
 1100 L$="   "+CHR$(224)+" "+CHR$(10)+
      STRING$(5,CHR$(8))
 1110 L$=L$+"  "+CHR$(224)+"  "+CHR$(10)+
      STRING$(6,CHR$(8))
 1120 L$=L$+"  "+CHR$(224)+CHR$(226)+"  "
 1130 R$=" "+CHR$(225)+"   "+CHR$(10)+
      STRING$(5,CHR$(8))
```

```
1140 R$=R$+"  "+CHR$(225)+CHR$(10)+
     STRING$(3,CHR$(8))

1150 R$=R$+"  "+CHR$(226)+CHR$(225)+" "
1200 M%=1
1210 S$=L$
1220 X%=23
1260 FROG=1
1270 GAME_END=FALSE
1280 SC=0
1290 ENDPROC

2000 DEF PROCprint_flies
2020 CLS
2030 C%=0
2040 FOR V=2 TO 10 STEP 2
2050 C%=NOT C%
2060 FOR U=2 TO 36 STEP 2
2070 PRINT TAB(U-C%,V);CHR$(227);
2080 NEXT U
2090 NEXT V
2100 ENDPROC

3000 DEF PROCmove_ss
3010 IF INKEY(-26) AND X%>0 THEN X%=X%-1
3020 IF INKEY(-122) AND X%<35 THEN X%=X%+1
3025 IF INKEY(-58) THEN PROCflip
3030 PRINT TAB(X%,20);S$;
3040 IF M%=0 THEN K%=X%+3:F$=CHR$(228)+" "
     ELSE K%=X%:F$=" "
     +CHR$(228)
3060 PRINT TAB(K%,21);F$;
3070 ENDPROC

4000 DEF PROCmove_f
4010 PRINT TAB(F,G);" ";
4020 F=F+B:G=G+A:A=A+P
4030 IF F>38 OR F<2 THEN B=-B
4040 IF G<2 THEN G=2:A=-A
4050 IF G<11 THEN PROChit_fly
4060 PRINT TAB(F,G);CHR$(228);
4070 IF G>20 THEN PROCseesaw
4080 ENDPROC

4500 DEF PROCseesaw
4510 D%=F-X%-M%
4520 IF D%<1 OR D%>2 THEN PROCmiss:ENDPROC
4525 IF M%=1 THEN B=ABS(B) ELSE B=-ABS(B)
```

```
4530 IF M%=1 THEN S$=R$:M%=0 ELSE S$=L$:M%=1
4540 A=-A
4550 G=20:F=K%+1
4560 SOUND 1,-15,200,1
4570 ENDPROC

4600 DEF PROCflip
4610 IF M%=1 THEN S$=R$:M%=0 ELSE S$=L$:M%=1
4620 TIME=0
4630 REPEAT
4640 UNTIL TIME>10
4650 ENDPROC


5000 DEF PROCstartup
5010 PROCmove_ss
5015 VDU 23,1,0;0;0;0;
5020 COLOUR 128+1
5030 PRINT TAB(35,15)STRING$(5," ");
5035 COLOUR 128+0
5040 FOR F=39 TO 34 STEP -.15
5050 PRINT TAB(F,14);CHR$(228);" ";
5060 PROCmove_ss
5070 NEXT F
5080 FOR G=14 TO 20 STEP .5
5090 PRINT TAB(34,G-1);" ";
5100 PRINT TAB(34,G);CHR$(228)
5110 PROCmove_ss
5120 NEXT G
5130 F=34
5140 G=20
5150 A=.4+RND(0)/10
5160 B=-(.1+RND(0)/2.5)
5165 P=0.008
5170 PRINT TAB(2,30);"FROG ";FROG;" SCORE=";SC
5180 A=A+A/D
5190 B=B+B/D
5200 P=P+P/D
5210 PROCseesaw
5220 ENDPROC


6000 DEF PROCtitle
6020 CLS
6040 PRINT TAB(10,5);"L E A P   F R O G"
```

```
6050 PRINT TAB(0,10);
6060 PRINT " In this game you must eat the
     flies ";CHR$(227)
6065 PRINT
6070 PRINT " by jumping your two frogs ";
     CHR$(228);" off the"
6075 PRINT
6080 PRINT " seesaw using the left and right
     arrow"
6085 PRINT
6090 PRINT " keys - you have ten frogs to clear"
6095 PRINT
6100 PRINT " the screen ...."
6110 PRINT TAB(5,25);"Difficulty level "
6120 PRINT TAB(8);"1 (difficult) to 10  (easy)";
6125 INPUT D
6130 IF D<1 OR D>10 THEN GOTO 6020
6140 ENDPROC

7000 DEF PROChit_fly
7010 IF FNC(F,G)<>1 THEN ENDPROC
7020 SOUND 1,-15,100,1
7030 SC=SC+1
7040 PRINT TAB(2,30);"FROG ";FROG;" SCORE=";SC
7050 IF SC=76 THEN GAME_END=TRUE
7060 ENDPROC


8000 DEF PROCmiss
8005 PRINT TAB(F,G);" ";
8010 FROG=FROG+1
8020 IF FROG>10 THEN GAME_END=TRUE:ENDPROC
8025 SOUND 1,-15,50,5
8030 IF M%=0 THEN M%=1:S$=L$
8040 PROCstartup
8050 ENDPROC

9000 DEF PROCend_game
9005 CLS
9020 PRINT TAB(1,10);
9030 IF SC<10 THEN PRINT "A mere tadpole could
     do better":GOTO 9400
9040 IF SC<20 THEN PRINT "Were your frogs
     three-legged?":GOTO 9400
```

```
9050 IF SC<40 THEN PRINT "Not bad for a
     shortsighted toad":GOTO 9400
9060 IF SC<50 THEN PRINT "Pretty good
     considering....":GOTO 9400
9070 IF SC<60 THEN PRINT "A great fly eating
     operation"
9080 PRINT "Try entering the frog olympics"
9400 PRINT TAB(5,15);"You scored ";SC
9410 PRINT TAB(10,18);
9420 INPUT "Another game (Y/N)",A$
9430 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 9400
9440 IF LEFT$(A$,1)="Y" THEN AGAIN=TRUE ELSE
     AGAIN=FALSE
9450 ENDPROC

9500 DEF FNC(X%,Y%)
9510 X%=16+32*X%
9520 Y%=1011-32*Y%
9530 =POINT(X%,Y%)
```

Chapter Four
Frogling

A frog plays a central role in this as well as in last chapter's game Frogling is a version of the classic arcade game of Frogger. The object of the game is to get a frog safely across a busy road, dodging oncoming traffic, and then across a fast flowing river, by jumping on floating logs The implementation of this game in BASIC provides an opportunity to explain and experiment with a type of animation that relies on scrolling.

Scrolling animation

Scrolling has now become a familiar part of text display on almost every computer. We are all used to the idea that when a text screen is full a subsequent PRINT statement will cause the whole screen to moved up by one line, thus losing the top line and freeing the bottom line. What is not so familiar is that scrolling can be useful as a method of animation. The problems associated with trying to print a large number of characters fast enough to make animation possible have already been encountered in the first two games in this book. With these problems in mind you should be able to see that a screen scroll is way of moving a large number of characters in a very short time an with very little programming effort. For example, try:

```
10 MODE 5
20 PRINT TAB(8,31);
40 PRINT TAB(RND(38);"*";
50 GOTO 20
```

This program prints asterisks at random positions at the bottom of the screen and animates them vertically up the screen using scrolls. As the program demonstrates it is possible to animate a large number of objects using scrolling animation. In fact, the problem in scrolling animation is keeping keeping things still! This can be done by printing

objects at the end of one scroll erasing them before the next.

Scrolling animation is a possible candidate whenever a game consists of a large number of objects moving at the same speed and in the same direction with perhaps one or two objects having a different pattern of motion. In the case of Frogling, the traffic on the road and the logs floating in the river constitute the large number of moving objects and the frog hopping hopping its way between the traffic and across the river is the single object that moves differently from the rest.

The game design

The idea of using scrolling animation to implement a version of Frogger implies that the traffic and the logs have to move up the screen and the frog horizontally across the screen. To make the game reasonably interesting to play there have to be at least two lanes of traffic and two columns of floating logs. Starting from the far left hand side of the screen, the player has to make the frog jump between the traffic and onto a safe strip between the road and the river. The river has to be crossed by jumping onto logs. Notice that while the frog is on the road it is stationary, apart from jumping under the control of the player, but while it is on a log it is swept up the river with the log. The frog has to reach the other bank before the log that it is sitting on sweeps it off the top of the screen. To increase the pressure on the player to move the frog there is also a time limit imposed. ALso, while the player can move the frog up and down the screen and to the right across the road and river, it cannot be moved to the left to retreat to safety once the frog starts moving forward it must continue! The layout of the game can be seen in Fig. 4.1.

The use of text windows

The only trouble with using scrolling to animate the traffic and the logs in frogling is that this method causes them all to move at the same rate and at the same time. This ‘all together’ movement makes the game rather too easy and boring to play. The problem of crossing the road, for example, comes down to waiting for a suitable gap to appear in the two lines of traffic and then hopping across as quickly as possible. Crossing the river is even easier. All you have to do is to wait for two logs to appear next to each other and then jump on the first log an then

immediately onto the second log.

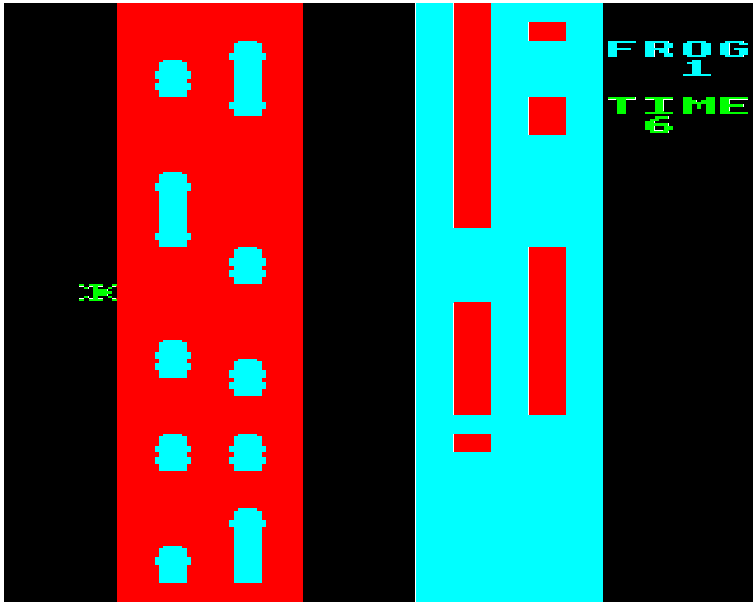


Fig. 4.1.

If the lines of traffic and logs all move at different rates then the game becomes much more interesting. For example, suppose the second lane of traffic moves twice as fast as the first. In this case any gaps in the lanes of traffic are not fixed in relation to one another – the gaps in the second lane continually overtake the gaps in the first lane and this makes the decision about when to make the frog jump much more difficult. The same is true if the first set of logs moves at twice the rate the second set. The frog can jump onto a log only to find that the second log that he was about to jump on has been left behind!

It is obvious that for a good game of Frogling we have to find some way of making the two lanes of traffic and the two lines of logs move at different rates. This sound like a difficult problem but in fact it is very easy using the Electron's text window facility. Using VDU 28 it is possible to restrict the area of the screen that is used for text output – i.e. to create a text window. From our point of view the most important

feature of a text window is that it behaves exactly like the whole screen including the way that it scrolls when you print on its bottom line. The idea is to use a one character wide by 32 hues deep text window for each lane of traffic and each set of logs and scroll them independently. For example, try running

```
10 MODE 5
20 VDU 28,4,31,4,0
30 END
```

and then list the program. Yon will find that after this program all the output is confined to a single column on the screen. When you have seen this program demonstrate a single column text window it is not difficult to imagine how it could be used to animate the traffic and frogs in Frogling.

Main program and PROCtitle

In this game the main program is a little different from the ones previously presented and contains some statements that are not simply procedure calls!

```
 10 REM Frogling
 20 MODE 5
 25 PROCtitle
 30 FOR FROG=1 TO 3
 40 PROCinit
 50 COLOUR 128
 60 PRINT TAB(16,2);"FROG";TAB(18,3);FROG
 70 TIME=0
 80 REPEAT
 90 PROCscroll
100 PROCmove_f
110 COLOUR 128
120 IF TIME-500>0 THEN PRINT TAB(16,5);"TIME";
    TAB(17,6);TIME DIV 100-5
130 UNTIL TIME-500>MAX OR GAME_END
140 PROCdeath
150 NEXT FROG
160 PROCend_game
170 IF AGAIN THEN RUN
180 END
```

Mode 5 (20 columns by 32 lines in four colours) is selected by line 20 Frogling certainly needs four colours but 20 columns is more than enough for two lanes of traffic and two sets of logs. PROCtitle simply prints the directions tor the game. The rest of the nmin program is in thr form of two nested loops. Lines 30 to 150 form a FOR loop that repeats the game three times with three different frogs. The inner loop, lines 80 to 130 is the animation loop. PROCinit, called at line 40, does the usual job of initialising everything that needs to be initialised. Lines 50 and 60 print the number of the current frog before the game proper begins, The animation loop calls PROCscroll to animate the road and the river and then calls PROCmove_f to allow the player to move the frog, Finally, line 120 prints the current time and line 130 checks for the end of the game. PROCdeath either signals the end of another frog or a success in getting to the other side. The whole game is brought to an end by PROCend_game which also asks if another game is desired. The only other point worth mentioning is the use of TIME-500 in the IF statement at line 120 and the UNTIL statement at line 130. This is because for the first five seconds that the animation loop is running the frog cannot move. This five second start-up is necessary to allow the cars and logs to scroll up the screen and make the game a challenge to play. So the starting sequence is that the road and river are drawn, the traffic and logs are animated up the screen for five seconds and then the frog appears and the timer starts.

PROCtitle is so simple that it is listed below without comment!

```
7000 DEF PROCtitle
7010 COLOUR 2
7020 COLOUR 128
7030 CLS
7040 PRINT TAB(2,2);"F R O G L I N G"
7050 PRINT TAB(1,5);"In this game you"'
7060 PRINT TAB(3);"have to guide"'
7070 PRINT TAB(1);"three frogs across"'
7080 PRINT TAB(2);"a busy road and a "'
7090 PRINT TAB(1);"fast flowing river."'
7100 PRINT TAB(1);"Use the up, down"'
7110 PRINT TAB(3);"and the right"'
7120 PRINT TAB(4);"arrow keys"'
7125 PRINT TAB(1);"Don't go under a"'
7130 PRINT TAB(2);"car and don't"
7140 PRINT TAB(1);"fall in the river."'
7150 PRINT TAB(3);"Press any key"
```

```
7155 PRINT TAB(4);" to start"
7160 IF INKEY$(0)="" THEN GOTO 7160
7170 ENDPROC
```

PROCinit

PROCinit starts off in the usual way by defining the graphics characters and colours used but in this case it is also responsible for printing the coloured strips that represent the road and the river.

```
1000 DEF PROCinit
1010 VDU 23,224,&3C,&7E,&7E,&7E,&7E,&FF,&FF,&FF
1020 VDU 23,225,&7E,&7E,&7E,&7E,&7E,&7E,&7E,&7E
1030 VDU 23,226,&7E,&7E,&FF,&FF,&FF,&7E,&7E,&7E
1040 VDU 23,227,&B9,&52,&1C,&1E,&1C,&52,&B9,&00
1050 VDU 23,228,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
1060 VDU 19,0,0,0,0,0,0:REM 0=BLACK
1070 VDU 19,1,1,0,0,0,0:REM 1=RED
1080 VDU 19,2,6,0,0,0,0:REM 2=CYAN
1090 VDU 19,3,2,0,0,0,0:REM 3=GREEN
1100 A%=32
1110 B%=32
1120 C%=32
1130 D%=32
1140 VDU 23,1,0;0;0;0;
1160 X%=2
1170 Y%=15
1175 VDU 26:CLS
1180 FOR I=0 TO 31
1190 COLOUR 128+1
1200 PRINT TAB(3,I);SPC(5);
1210 COLOUR 128+2
1220 PRINT TAB(11,I);SPC(5);
1230 NEXT I
1240 GAME_END=FALSE
1250 MAX=1000
1260 ENDPROC
```

Lines 1010 to 1050 define the cars, logs and frog. CHR$(224) is a car or lorry front, CHR$(225) is a lorry middle section and CHR$(226) is a car or lorry end. You can see the way that these three characters go together in Fig. 4.2. CHR$(227) is the frog and CHR$(228) is simply a solid block used to make up logs (see Fig. 4.2). The colours selected by lines 1060 to 1090 are black for the background, red for the road and for the logs, blue for the river and the traffic and green for the frog. You

may be surprised at the choice of bue cars on a red road and red logs on a blue river but this does simplify the game quite a lot. When the frog is crossing the road it must avoid blue cars and when it is crossing the river it must avoid blue water and so all through the game the colour blue indicates an area where the frog shouldn't go.

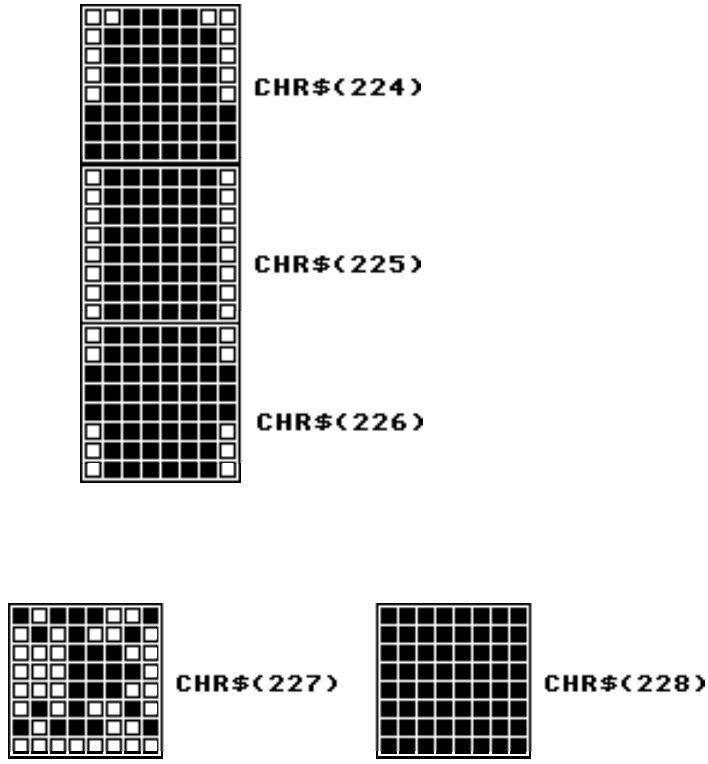


Fig. 4.2. Graphics characters for lorry, frog and log

Lines 1100 to 1130 initialise the variables A% D%, to 32, the ASCII code for space. These variables are used by PROCscroll and are best described in that section. X% and Y%, are the current co-ordinates of the frog and are initialised by lines 1160 and 1170.

The VDU 26:CLS command in line 1175 removes any text windows already defined and clears the screen. Lines 1180 to 1230 print two bands of colour down the screen. The red band at the left hand side of

the screen represents the road and the blue band to the right represents the river. Finally, line 1240 sets the 'game end' flag to false and line 1250 sets the time limit for the game to ten seconds.

PROCscroll and associated procedures

The implementation of the program falls neatly into two areas the generation and scrolling of the road and river, and the movement of the frog. The problems of animating the frog are dealt with later. The animation of the road and river is dealt with by PROCscroll and the procedures that it calls and this is the subject of this section.

```
3000 DEF PROCscroll
3010 A%=FNgen_car(.4,.5,A%)
3020 PROCp_wind(1,A%)
3030 B%=FNgen_car(.16,.5,B%)
3040 PROCp_wind(2,B%)
3045 B%=FNgen_car(.2,.5,B%)
3046 PROCp_wind(5,B%)
3050 C%=FNgen_log(.16,.2,C%)
3060 PROCp_wind(3,C%)
3070 C%=FNgen_log(.1,.2,C%)
3080 PROCp_wind(5,C%)
3090 D%=FNgen_log(.1,.2,D%)
3100 PROCp_wind(4,D%)
3110 PROCp_wind(6,0)
3120 ENDPROC
```

At first sight PROCscroll looks complicated but this is only because the purpose of the functions FNgen_car and FNgen_log and the procedure PROCp_wind is not immediately obvious. To understand what is going on you have to think about what is required at each step of the animation. Consider, for example, the first lane of the road. At each animation step we have to decide whether to print a blank, the start of a car or lorry, a middle section of a lorry, or the end of a car or lorry. This is what FNgen_car(PSTART,PEND,OB%) does each time it is called.

PSTART is the probability that a car or lorry will be generated – that is, it controls the number of cars and lorries on the road. PEND is the probability that a car or lorry will come to and end that is, it controls the number of cars as opposed to lorries and the average length of the lorries. (A car is simply a car front followed by a car end and a lorry is a car front followed by a number of lorry middle characters and then a

car end – see Fig. 4.2.) OB% is simply the ASCII code of the last character that was printed in the lane. For example, if the last character was a space (ASCII code 32) then the next character to be printed can either be another space or a car front (it clearly cannot be a lorry middle section or a car end). Whether the next character is a space or a car front is decided at random with a probability determined by PSTART.

In PROCscroll A% is used to hold the character printed in traffic lane one and line 3010 calls FNgen_car to update it. FNgen_car returns the character code of the character to be printed next. In the same way B% is used to hold the character printed in traffic lane two, and C% and D% hold the character codes to be printed in the first and second column of logs respectively. FNgen_log performs the same action as FNgen_car but for the river that is, it returns the code of the next character to be printed, given the code of the last character that was printed. The action of FNgen_car and FNgen_log will become clearer after they have been described in more detail later.

PROCp_wind(N%,C%) is the procedure that actually does the printing and scrolling. N% controls which of the four text windows the character will be printed in and C% is the ASCII code of the character to be printed.

You should now be able to understand the structure of PROCscroli. Line 3010 calls FNgen_car to generate the character to be printed in the first lane of traffic. Then line 3020 calls PROCp_wind to print it in window one. In the same way lines 3030 to 3046 generate and print two car characters for window two. As window two is scrolled twice each time window one is scrolled once it appears to move twice as fast. Lines 3050 to 3070 generate and print two log characters in text window three and lines 3090 and 3100 generate and print one log character in text window four. As text window three is scrolled twice each time text window four is scrolled, the first set of logs moves twice as fast as the second set. Finally, line 3110 calls PROCp_wind to set the text window to the whole of the screen.

As already mentioned, to understand fully how PROCscroll works you have to understand the functions and procedures that it calls. The first of these is FNgen_car:

```
2000 DEF FNgen_car(PSTART,PEND,OB%)
2010 LOCAL T%
2015 T%=OB%
2020 IF OB%=32 AND RND(1)<PSTART THEN T%=224
2030 IF OB%=224 THEN T%=225
2040 IF OB%=226 THEN T%=32
```

```
2050 IF T%=225 AND RND(1)<PEND THEN T%=226
2060 =T%
```

Line 2020 tests to see if the last character printed was a blank. If it was, then the next character to be printed will either be another blank or the front of a car. If the last character was the front of a car then line 2030 makes the next character the middle section of a lorry. Line 2040 checks to see if the last character was a car end and if it was, the next character will be a blank. Finally, line 2050 changes lorry middle sections to car ends at random.

FNgen_log is simpler than FNgen_car:

```
5000 DEF FNgen_log(PSTART,PEND,OB%)
5020 IF OB%=32 AND RND(1)<PSTART THEN =228
5030 IF OB%=228 AND RND(1)<PEND THEN =32
5040 =OB%
```

If the last character was a space then line 5020 will make the next character a log with probability PST ART. Line 5030 does the reverse and a log character into a space with probability PEND. You should be able to see that PST ART is the probability of starting a log and PEND is the probability of finishing a log.

PROCp_wind sets up text windows and prints the next character:

```
4000 DEF PROCp_wind(N%,C%)
4010 IF N%=1 THEN VDU 28,4,31,4,0:
     VDU 17,2,17,129
4020 IF N%=2 THEN VDU 28,6,31,6,0
4030 IF N%=3 THEN VDU 28,12,31,12,0:
     VDU 17,1,17,130
4040 IF N%=4 THEN VDU 28,14,31,14,0:VDU 17,1
4050 IF N%=6 THEN VDU 28,0,31,19,0:ENDPROC
4100 PRINT TAB(0,31);CHR$(C%);
4130 ENDPROC
```

If N% is 1, line 4010 sets up text window one and also sets the foreground colour to blue and the background colour to red (text window one is part of the road). In the same way lines 4020. 4030 and 4040 sets up text windows two, three and four respectively. When N% is 5 a new text window is not set up and as a result the text window set up by the last call remains in use. When N% is 6 line 4050 restores the text window to the entire screen and then returns to the calling

procedure. Line 4100 prints the character whose code is in C% at the bottom of the current text window and so causes it to scroll.

PROCmove_f and PROCover

On the face of it PROCmove_f has a very simple job to do and should be very familiar to all other procedures that move objects around the screen. However PROCmove_f has a number of difficulties to overcome. In particular, the background colour used to blank and print the frog has to be changed according to whether the frog is on the road, on a log or on neither. In addition, the way that the frog moves depends on whether it is within one of the scrolling text windows or not. For example, if it is within the first text window (i.e. the first lane of the road) it will be scrolled up along with the rest of the traffic. To keep it in the same place it is necessary to blank out the frog at its old position and reprint it. But if the frog is within the third or forth text windows then, as long as it is sitting on a log, it should be scrolled upward along with the rest of the logs.

```
6000 DEF PROCmove_f
6005 IF TIME<500 THEN ENDPROC
6010 COLOUR 3
6016 IF X%=4 OR X%=6 THEN K%=X% DIV 2-1
     ELSE K%=0
6018 COLOUR 128+FNC(X%,Y%-K%)
6019 PRINT TAB(X%,Y%-K%);" ";
6020 IF INKEY(-58) AND Y%>0 THEN Y%=Y%-1
6030 IF INKEY(-42) AND Y%<31 THEN Y%=Y%+1
6040 IF INKEY(-122) AND X%<16 THEN
     SOUND 1,1,30,2:SOUND 1,1,40,2:X%=X%+2
6045 IF X%=8 OR X%=16 THEN X%=X%+2
6046 Z%=FNC(X%,Y%)
6047 IF Z%=2 THEN GAME_END=TRUE
6049 COLOUR 128+Z%
6050 PRINT TAB(X%,Y%);CHR$(227);
6055 IF X%=12 OR X%=14 THEN PROCover
6056 IF X%=12 THEN PROCover
6060 IF X%>14 THEN GAME_END=TRUE
6070 ENDPROC
```

Line 6005 stops the frog from moving until five seconds have elapsed since the start of the game. Line 6016 works out where a space should

be printed to blank out the frog at its old position. If the frog is within either text window one (X=4) or text window two (X%=6) then it will have been scrolled up by either two lines or one line respectively.

Before the frog is blanked out line 6018 sets the background colour to be the same as the colour already displayed at the location. This is achieved simply by use of FNC(X%,Y%) which was first introduced in Ant Hill and returns the colour of the pixel at X%,Y%. Line 6019 finally prints the blanking space at the correct position and in the correct colour. Lines 6020 to 6040 alter the frog's co-ordinates depending on which arrow key is pressed. Line 6045 automatically makes the frog perform a double jump to take it to the left hand bank of the river or away from the right hand bank. After this line the new position of the frog is in X%,Y%.

Line 6046 finds the colour of the character location that the frog is about to jump on and stores it in Z%. If the colour is blue, i.e. colour code 2, then line 6047 sets GAME_END to TRUE. Otherwise line 6049 sets the background colour to the same colour that is already present in X%,Y% and the line 6050 prints the frog.

Line 6055 tests to see if the frog is in either of text windows three and four and if this is the case PROCover is called both to update the frog's y co-ordinate as a result of it being scrolled up and to test if the frog has reached the top of the screen. Line 6056 calls PROCover a second time if the frog is in text window three to allow for the fact that text window three is scrolled twice. Finally line 6060 tests to see if the frog has safely reached the far side of the river.

It only remains to give the listings of PROCover and FNC:

```
6500 DEF PROCover
6510 IF Y%=0 THEN GAME_END=TRUE:ENDPROC
6520 Y%=Y%-1
6540 ENDPROC

9000 DEF FNC(X%,Y%)
9010 X%=16+32*X%
9020 Y%=1011-32*Y%
9030 =POINT(X%,Y%)
```

PROCdeath and PROCend_game

The final procedures in the game, PROCdeath and PROCend_game, are very simple. PROCdeath tests to find out whether the frog made its journey successfully and if not, how it met its end, and sounds a note

accordingly. PROCend_game asks the player if another game is required.

```
6600 DEF PROCdeath
6610 IF X%>15 THEN SOUND 1,-15,100,20 ELSE
      SOUND 1,-15,50,10
6620 TIME=0
6630 REPEAT
6640 UNTIL TIME>100
6650 ENDPROC

8000 DEF PROCend_game
8010 CLS
8020 PRINT TAB(1,10);"Another game";
8030 INPUT A$
8040 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 8010
8050 IF LEFT$(A$,1="Y" THEN AGAIN=TRUE ELSE
     AGAIN=FALSE
8060 ENDPROC
```

Evaluation and improvements

Frogling certainly runs fast enough to be interesting and the scrolling animation used in combination with text windows produces an interesting display at the cost of very little programming. Perhaps the most disappointing aspect of the game, however, is its dull sound effects. This is soon put to rights with the addition of an ENVELOPE statement to PROCinit:

```
1260 ENVELOPE 1,1,1,-2,1,2,2,2,
     126,0,0,-126,126,1
1270 ENDPROC
```

This produces a sound with a rapidly rising and falling pitch but there is still plenty of scope to alter and adjust the sound using the SOUND command. Using this envelope PROCdeath becomes:

```
6600 DEF PROCdeath
6610 IF X%=12 OR X%=14 THEN SOUND 0,-15,4,5
6615 IF X%=4 OR X%=6 THEN SOUND 1,1,10,20
6616 IF X%<15 THEN GOTO 6620
6617 FOR I=50 TO 100 STEP 8
6618 SOUND 1,1,I,4
6619 NEXT I
6620 TIME=0
```

```
6630 REPEAT
6640 UNTIL TIME>100
6650 ENDPROC
```

The same envelope can also be used to make a more appropriate sound when the frog jumps. Change line 6040 to

```
6040 IF INKEY(-122) AND X%<16 THEN
     SOUND 1,1,30,2:SOUND 1,1,40,2:X%=X%+2
```

The addition of these simple sound effects certainly increases both the tension and the enjoyment of the game and you should try playing the game both with and without them. Even though this is the point at which we left Frogling, this doesn't mean that there isn't still more scope for extending the game. In particular you could add a scoring system based on the number of frogs guided safely to the other side and the time taken and then alter PROCend_game to print appropriate comments along the lines of those found in Leap Frog

The final version – a complete listing

```
  10 REM Frogling
  20 MODE 5
  25 PROCtitle
  30 FOR FROG=1 TO 3
  40 PROCinit
  50 COLOUR 128
  60 PRINT TAB(16,2);"FROG";TAB(18,3);FROG
  70 TIME=0
  80 REPEAT
  90 PROCscroll
 100 PROCmove_f
 110 COLOUR 128
 120 IF TIME-500>0 THEN PRINT TAB(16,5);"TIME";
     TAB(17,6);TIME DIV 100-5
 130 UNTIL TIME-500>MAX OR GAME_END
 140 PROCdeath
 150 NEXT FROG
 160 PROCend_game
 170 IF AGAIN THEN RUN
 180 END
```

```
1000 DEF PROCinit
1010 VDU 23,224,&3C,&7E,&7E,&7E,&7E,&FF,&FF,&FF
1020 VDU 23,225,&7E,&7E,&7E,&7E,&7E,&7E,&7E,&7E
1030 VDU 23,226,&7E,&7E,&FF,&FF,&FF,&7E,&7E,&7E
1040 VDU 23,227,&B9,&52,&1C,&1E,&1C,&52,&B9,&00
1050 VDU 23,228,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
1060 VDU 19,0,0,0,0,0,0:REM 0=BLACK
1070 VDU 19,1,1,0,0,0,0:REM 1=RED
1080 VDU 19,2,6,0,0,0,0:REM 2=CYAN
1090 VDU 19,3,2,0,0,0,0:REM 3=GREEN
1100 A%=32
1110 B%=32
1120 C%=32
1130 D%=32
1140 VDU 23,1,0;0;0;0;
1160 X%=2
1170 Y%=15
1175 VDU 26:CLS
1180 FOR I=0 TO 31
1190 COLOUR 128+1
1200 PRINT TAB(3,I);SPC(5);
1210 COLOUR 128+2
1220 PRINT TAB(11,I);SPC(5);
1230 NEXT I
1240 GAME_END=FALSE
1250 MAX=1000
1260 ENVELOPE 1,1,1,-2,1,2,2,2,
     126,0,0,-126,126,1
1270 ENDPROC


2000 DEF FNgen_car(PSTART,PEND,OB%)
2010 LOCAL T%
2015 T%=OB%
2020 IF OB%=32 AND RND(1)<PSTART THEN T%=224
2030 IF OB%=224 THEN T%=225
2040 IF OB%=226 THEN T%=32
2050 IF T%=225 AND RND(1)<PEND THEN T%=226
2060 =T%


3000 DEF PROCscroll
3010 A%=FNgen_car(.4,.5,A%)
3020 PROCp_wind(1,A%)
3030 B%=FNgen_car(.16,.5,B%)
3040 PROCp_wind(2,B%)
3045 B%=FNgen_car(.2,.5,B%)
3046 PROCp_wind(5,B%)
```

```
3050 C%=FNgen_log(.16,.2,C%)
3060 PROCp_wind(3,C%)
3070 C%=FNgen_log(.1,.2,C%)
3080 PROCp_wind(5,C%)
3090 D%=FNgen_log(.1,.2,D%)
3100 PROCp_wind(4,D%)
3110 PROCp_wind(6,0)
3120 ENDPROC

4000 DEF PROCp_wind(N%,C%)
4010 IF N%=1 THEN VDU 28,4,31,4,0:
     VDU 17,2,17,129
4020 IF N%=2 THEN VDU 28,6,31,6,0
4030 IF N%=3 THEN VDU 28,12,31,12,0:
     VDU 17,1,17,130
4040 IF N%=4 THEN VDU 28,14,31,14,0:VDU 17,1
4050 IF N%=6 THEN VDU 28,0,31,19,0:ENDPROC
4100 PRINT TAB(0,31);CHR$(C%);
4130 ENDPROC

5000 DEF FNgen_log(PSTART,PEND,OB%)
5020 IF OB%=32 AND RND(1)<PSTART THEN =228
5030 IF OB%=228 AND RND(1)<PEND THEN =32
5040 =OB%

6000 DEF PROCmove_f
6005 IF TIME<500 THEN ENDPROC
6010 COLOUR 3
6016 IF X%=4 OR X%=6 THEN K%=X% DIV 2-1
     ELSE K%=0
6018 COLOUR 128+FNC(X%,Y%-K%)
6019 PRINT TAB(X%,Y%-K%);" ";
6020 IF INKEY(-58) AND Y%>0 THEN Y%=Y%-1
6030 IF INKEY(-42) AND Y%<31 THEN Y%=Y%+1
6040 IF INKEY(-122) AND X%<16 THEN
     SOUND 1,1,30,2:SOUND 1,1,40,2:X%=X%+2
6045 IF X%=8 OR X%=16 THEN X%=X%+2
6046 Z%=FNC(X%,Y%)
6047 IF Z%=2 THEN GAME_END=TRUE
6049 COLOUR 128+Z%
6050 PRINT TAB(X%,Y%);CHR$(227);
6055 IF X%=12 OR X%=14 THEN PROCover
6056 IF X%=12 THEN PROCover
6060 IF X%>14 THEN GAME_END=TRUE
6070 ENDPROC

6500 DEF PROCover
6510 IF Y%=0 THEN GAME_END=TRUE:ENDPROC
```

```
6520 Y%=Y%-1
6540 ENDPROC

6600 DEF PROCdeath
6610 IF X%=12 OR X%=14 THEN SOUND 0,-15,4,5
6615 IF X%=4 OR X%=6 THEN SOUND 1,1,10,20
6616 IF X%<15 THEN GOTO 6620
6617 FOR I=50 TO 100 STEP 8
6618 SOUND 1,1,I,4
6619 NEXT I
6620 TIME=0
6630 REPEAT
6640 UNTIL TIME>100
6650 ENDPROC

7000 DEF PROCtitle
7010 COLOUR 2
7020 COLOUR 128
7030 CLS
7040 PRINT TAB(2,2);"F R O G L I N G"
7050 PRINT TAB(1,5);"In this game you"'
7060 PRINT TAB(3);"have to guide"'
7070 PRINT TAB(1);"three frogs across"'
7080 PRINT TAB(2);"a busy road and a "'
7090 PRINT TAB(1);"fast flowing river."'
7100 PRINT TAB(1);"Use the up, down"'
7110 PRINT TAB(3);"and the right"'
7120 PRINT TAB(4);"arrow keys"'
7125 PRINT TAB(1);"Don't go under a"'
7130 PRINT TAB(2);"car and don't"
7140 PRINT TAB(1);"fall in the river."'
7150 PRINT TAB(3);"Press any key"
7155 PRINT TAB(4);" to start"
7160 IF INKEY$(0)="" THEN GOTO 7160
7170 ENDPROC

8000 DEF PROCend_game
8010 CLS
8020 PRINT TAB(1,10);"Another game";
8030 INPUT A$
8040 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 8010
8050 IF LEFT$(Aame (Y/N)",A$
9430 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 9400
9440 IF LEFT$(A$,1)="Y" THEN AGAIN=TRUE ELSE
     AGAIN=FALSE
9450 ENDPROC
```

```
9000 DEF FNC(X%,Y%)
9010 X%=16+32*X%
9020 Y%=1011-32*Y%
9030 =POINT(X%,Y%)
```

Chapter Five
Snake

Animating a large object is normally a very difficult problem no matter what language you use. If you try to use BASIC then the problem lies in trying to alter a large number of pixels fast enough to give the impression of smooth motion. On the other hand if you use assembler then the main problem lies in implementing the complicated calculations that keep track of where everything is. Put simply, the trouble with trying to make a large object move is that it usually Involves printing, blanking and then reprinting too many characters each time through the animation loop.

There is one large object, however, that can easily be animated at speed and also has a complicated and fascinating way of moving – a snake. A snake can be made to wiggle its way around the screen in such a way that its speed doesn't depend on its length. If the direction of .movement of the snake is controlled by the four arrow keys then there is something compulsive about 'driving' it around the screen and, with the addition of a few simple rules such as not being allowed to cross its own tail, the task immediately becomes a challenge of skill. In this chapter the fundamental method of snake animation is described and perhaps the most addictive game in this book is developed.

Snake animation

A snake is made up of a number of graphics characters printed in a line, One end of the line is referred to as the head and the other the tail. The direction in which a snake moves is essentially governed by the direction in which the head moves. Every other character that makes up the snake will eventually follow the movement of the head. For example, the second character in the snake follows the head one move behind. In other words, when the head moves to a new position the second character moves into the head's old position. In the same way

the third characters move into the second's old position and so on all the way down the snake to the tail.

This description of how a snake moves is accurate but it makes it sound as if all the characters in the snake actually move each time the head does. If this were the case, animating a snake of any size would very quickly become a problem for assembly language. However, if all the characters that make up the snake are the same, only two characters – the head and the tail – actually need to move to give the impression that the whole snake is moving. The reason for this is not difficult to see. The head has to move because it is moving into a character location that was previously blank (or more generally, just not part of the snake). On the other hand the second character is moving into the character location that the head occupied and as they are the same character there is no visible change produced by the move. Obviously if we arrange not to blank the head's old position when it moves to its new position there is no need to reprint the second character at its new position! The argument can be applied to each character further down the snake until we reach the tail. In this case the argument about not having to move the tail into the position occupied by the last but one character holds but the old position of the tail does have to be blanked out because there is no snake character to move into its old position.

To summarise, if a snake is composed of identical characters it can be made to move by simply printing the head in its new position and blanking out the old position of the tail.

To see a simple demonstration of this method try the following program which animates a short snake composed of the letter 'S' across the screen in a fixed direction.

```
 10 MODE 6
 20 XH=10:YH=15
 30 XT=0:YT=15
 40 PRINT TAB(XH,YH);"S";
 50 PRINT TAB(XT,YT);" ";
 60 XH=XH+1
 70 XT=XT+1
 80 IF XH>39 THEN XH=1
 90 IF XT>39 THEN YT=1
100 TIME=0:REPEAT UNTIL TIME>5
110 GOTO 80
```

Line 20 defines the position of the head at XH, YH and line 30 sets the old position of the tail to be XT, YT. Line 40 prints the head at its new

position and line 50 blanks nut the old position of the tail. Lines 60 to 90 update the head and tail positions and then, after a short delay (line 100), the whole loop is repeated. Notice that what makes this example simple is that there is an easy way of updating the head and tail positions because the snake is moving horizontally.

In practice it is usual for the characters that form the head and tail of a snake to be different from the rest of the body but even this amendment causes very little in the way of extra work. If the head is different from the rest of the body then its old position has to in: changed to a character that forms the main body of the snake. If the tail is different then, as well as blanking out its old position, we also have to print the tail character at its new position. So even a 'good looking' snake with a clear head and tail only needs four characters printing to make it move, no matter how long it is.

The directional snake – the queue

Although the animation of a snake only requires the printing of a small number of characters each time through the animation loop, there is still a problem in keeping track of the positions of all the characters in the snake. As moving the snake only involves the head and the tail you might be puzzled as to why you need to keep track of the positions of all of the characters in the snake. The reason for this is the need to know where the tail will be printed at each move. When the snake is moving in a straight line, as in the example at the end of the last section, it is easy to know where the tail should move to. However, when the direction in which the head moves changes, it is altogether different. Each time the snake moves, the tail moves to the position that was occupied by tl last but one character and so it is clearly necessary always to know the position of the last but one character. But this argument can be repeated because each time the snake moves, the last but two characters becomes the last but one character in the snake. In other words, if all the co-ordinates of the Ith character in the snake are stored in X(I) and Y(I) then at each move, the co-ordinates are updated as follows:

```
XT=X(1)
YT=Y(1)

FOR I=N TO 2 STEP -1
X(I-1)=X(I)
Y(I-1)=Y(I)
NEXT I
```

and

```
X(N)=XH
Y(N)=YH
```

where the snake is N characters long, the co-ordinates of the head are stored in X(N) and Y(N), XT and YT are the co-ordinates of the old position of the tail and XH and YH are the new co-ordinates of the head. If you examine this FOR loop you should be able to see that it moves all the co-ordinates down the array by one place, the co-ordinates of the first character in the snake becoming the co-ordinates a the second character and so on. Following this shifting, the animation of the snake is achieved by simply printing the head at its new position:

```
PRINT TAB(X(N),Y(N));H$;
```

then changing the character at the head's old position to a snake 'body' character:

```
PRINT TAB(X(N-l),Y(N-l));S$;
```

and finally the tail is blanked out and then printed at its new position:

```
PRINT TAB(XT,YT);" ";
PRINT TAB(X(1),Y(l));T$;
```

where H$, S$ and T$ are strings containing the character used for the head, the body and the tail of the snake respectively.

The only trouble with the above method is that each time the snake moves, the contents of the pair of arrays X(l) and Y(1) have to be shifted down. This is quite a lot of work for a BASIC program to do each time through the animation loop and, what is worse, the amount of work increases with the length of the snake. Using this method in BASIC a a snake would move slowly and would grind (or slither?) to a virtual standstill as it grew in length. The solution to this difficulty is to be found in the use of an advanced data structure known as a queue. (If you would like to know more about the theory that lies behind data structures in general and the queue in particular then see Advanced Programming for the Electron, by Mike James (Granada, 1984) and also his The Complete Programmer (Granada, l983).) The basic idea is to avoid moving the data in the X and Y arrays by using a pair of pointers, one to the co-ordinates of the head and one to the co-ordinates of the tail. For example, if the co-ordinates of each character are once again stored in the arrays X and Y, with Q being the index of the array elements that hold the co-ordinates of the head and Z being the index of

the array elements that hold the co-ordinates of the tail, then the updating procedure becomes

```
Q=Q+1
X(Q) XH
Y(Q)=YH

Z=Z+1
```

In other words, the new position of the head of the snake is stored one element further up the array than its old position. in this way, each time the head moves it leaves behind it a trail of the co-ordinates of its old positions. So, for example, if the head is currently at X(Q), Y(Q) then its previous position was X(Q–1),Y(Q–1), and before that it was at X(Q–2), Y(Q–2) and so on. This trail of co-ordinates can be used to make the tail follow the head around the screen simply by mzving it from X(Z), Y(Z) to X(Z+1), Y(Z+1) each time through the animation loop. Technically the trail of co-ordinates and the variables Q and Z arc referred to as a queue. The term queue seems appropriate if you think of the co-ordinates of the head as joining the end of a queue of coordinates and the co-ordinates of the new position of the tail forming the front of the queue. As the snake moves forward co-ordinates move down the queue to eventually become the current position of the tail (see Fig. 5.1).

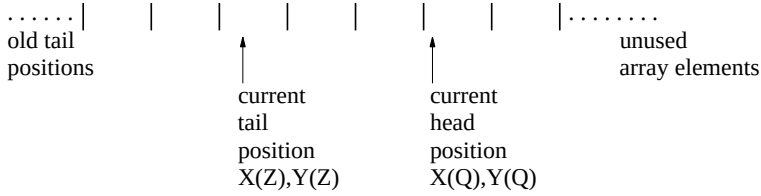


Fig. 5.1.

This method of storing the co-ordinates of the head has one big problem – as the snake moves around the screen, more and more array space is used up. However at any one time only the elements between X(Z), Y(Z) and X(Q), Y(Q) are needed. The rest of the arrays either are unused or hold now unwanted old tail positions. The solution is simply to make the array big enough to hold all of the co-ordinates of the longest snake that you are going to animate and if either of the pointers Z or Q reaches the limit of the arrays then reset them to I. The best way to imagine the way that this works is to think of the arrays as being circular, with their last element next to their first. In this sense the head

and tail pointers Q and Z move round in a circle with all the co-ordinates of interest stored between them. With this small addition we have all the ideas necessary to implement a range of games based on animated snakes.

The game design

There are many different possibilities for using animated snakes as part of a game but for the game featured in this chapter simplicity is the main objective. The basic idea is that the player has to guide a continuously moving snake around the screen with the aim of eating as many frogs as possible. As each frog is eaten the snake grows one characters longer and so a little more difficult and exciting to control the difficulty of the game further, the snake must not only avoid running into itself but must also avoid eating poisonous toads that hide among the frogs. Figure 5.2 shows a typical screen layout during the game. Other details of the game will be introduced as the program is described.

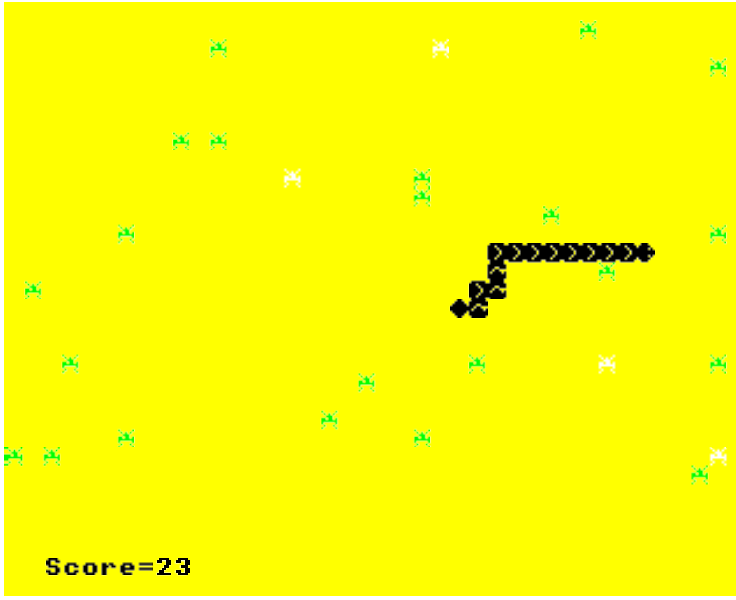


Fig. 5.2.

The main program

Once again the best choice of graphics is the four colour Mode 1. This gives just enough colours for a yellow background, black and yellow snake, green frogs and white poisonous toads. The 40 by 32 line screen also provides sufficient space for a fairly large snake to manoeuvre. The main program for this game follows the usual form:

```
 10 REM SNAKE
 20 MODE 1
 30 AGAIN=FALSE
 40 PROCinitialise
 50 PROCtitle
 60 PROCdraw_scene
 70 REPEAT
 80 PROCone_move
 90 UNTIL NO_FOOD OR NO_LIVES
100 PROCend_game
110 IF AGAIN THEN GOTO 40
120 END
```

Line 30 sets AGAIN to FALSE. This variable is used to indicate whether the program is being run for the first time or if a subsequent game is being played. Line 110 tests AGAIN and loops back if another game has been requested. Line 40 calls PROCinitialise to set up user defined characters, etc. Line 50 calls PROCtitle to display the title frame and then line 60 calls PROCdraw_scene to draw the frogs and toads and the snake in its starting position. Lines 70,80 and 90 form the animation loop. PROC one_move is responsible for moving the snake and checking to see if it has hit anything. Line 90 tests for the end of the game. The variable NO_FOOD) is true if the snake has managed to eat all of the frogs without using up its five fives and NO_L1VES is true if the snake has either run into itself or into a poisonous toad five times and hence used up its lives. Finally PROCend_game finishes the game with a suitable message and, if appropriate, allows the player to enter his or her name in a ‘best score’ table.

PROCinitialise

The user-defined characters that are used to make up the snake are a little more complicated than you might expect. As well as four different heads, CHR$(224), CHR$(227). CHR$(230) and CHR$(232), one for

each direction of travel, there are also four different snake body characters, CHR$(225), CHR$(228), CHR$(231) and CHR$(233) (see Fig. 5.3). By selecting a body character that points in the same direction as the head of the snake to overrprint the head at each move, the snake can be made to look even more impressive and full of movement. There is only one character used for the tail, CHRS(226), because the diamond shape looks the same in every direction of travel. The frog character is defined as CHR$(229) and this serves for both frog and poisonous toad. Frogs are CHR$(229) printed in green and toads are CHRS(229) printed in white.

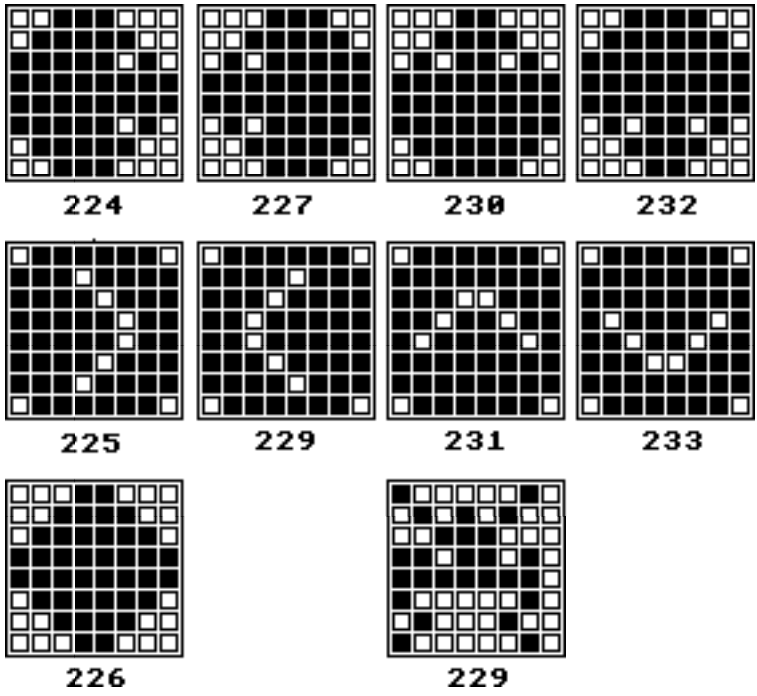


Fig. 5.3. Graphics characters for snake's head, snake's body, snake's tall and frog.

```
1000 DEF PROCinitialise
1010 VDU 23,224,&38,&7C,&FA,&FF,&FF,&FA,&7C,&38
1020 VDU 23,225,&7E,&EF,&F7,&FB,&FB,&F7,&EF,&7E
1030 VDU 23,226,&18,&3C,&7E,&FF,&FF,&7E,&3C,&18
1040 VDU 23,227,&1C,&3E,&5F,&FF,&FF,&5F,&3F,&1C
1050 VDU 23,228,&7E,&F7,&EF,&DF,&DF,&EF,&F7,&7E
1060 VDU 23,229,&82,&54,&38,&DA,&FE,&82,&44,&82
1070 VDU 23,230,&18,&3C,&5A,&FF,&FF,&FF,&7E,&7E
1080 VDU 23,231,&7E,&FF,&E7,&D8,&BD,&FF,&FF,&7E
1090 VDU 23,232,&3C,&7E,&FF,&FF,&FF,&5A,&3C,&18
1100 VDU 23,233,&7E,&FF,&FF,&BD,&DB,&E7,&FF,&7E
1110 VDU 19,0,3,0,0,0:REM 0=YELLOW
1120 VDU 19,1,0,0,0,0:REM 1=BLACK
1130 VDU 19,2,2,0,0,0:REM 2=GREEN
1140 VDU 19,3,7,0,0,0:REM 3=WHITE
1150 IF AGAIN THEN GOTO 1260
1200 DIM X%(50),Y%(50)
1205 MAX%=50
1210 DIM T$(5)
1220 DIM T(5)
1230 FOR I=1 TO 5
1240 T(I)=0
1250 NEXT I
1260 NO_FOOD=FALSE
1270 NO_LIVES=FALSE
1280 HIT=FALSE
1290 C%=0:STOTAL%=0
1300 SC=0:SN=0:FOOD=0
1310 ENDPROC
```

The first part of PROCinitialise sets up the user-defined character (lines 1010 to 1100) and then defines the logical to physical colour assignment (lines 1110 to 1140). Line 1150 skips the section of the procedure that defines arrays etc. if this isn't the first time thatt procedure has been called. This is necessary because in Acorn BASIC defining the same arrays twice in one run causes an error. In fact the character definitions and the logical to physical colour assignment could also be skipped after the first time the procedure is used, but the time lost through not doing so is minimal.

Line 1200 sets up the two arrays X% and Y% that will be used later in he program to store the snake's co-ordinates. MAX% (line 1205) is used to let the rest of the program know that the arrays are 50 elements long, hence limiting the maximum length of a snake. T$ and T, defined in lines 1210 and 1220, are used later in the program to hold

a league table of highest scores. Finallv, the last part of the procedure initialioses some general variables used later in the program. Lines 1260 and 1270 set the game indicators to FALSE. The variable HIT is TRUE if the snake has hit anything and FALSE otherwise. The variables C% and STOTAL% are used to keep the score as the game progresses. Each player is given five screens of frogs to eat and five lives to do it with! The variable C% records the number of frogs eaten from the current screenful and STOTAL% records the number of frogs eaten from previous screens. The variable SC is the number of the current screenful of frogs and SN is the number of the current life. The variable FOOD is used to keep track of the number of frogs left to eat.

The title frame

The procedure PROCtitle is fairly simple apart from its use of the snake characters to write the word SNAKE in large letters across the screen.

```
8000 DEF PROCtitle
8005 COLOUR 128+1:COLOUR 0
8006 CLS
8007 VDU 23,1,0;0;0;0;
8010 PROCsnake(5)
8020 PRINT TAB(2,10);
8030 PRINT "You must guide your snake using
     the"
8040 PRINT "four arrow keys and eat all the
     green"
8050 PRINT "frogs ";
8060 COLOUR 2:PRINT CHR$(229)
8070 COLOUR 0
8080 PRINT
8090 PRINT " If you try to eat a toad ";
8100 COLOUR 3:PRINT CHR$(229);
8110 COLOUR 0
8120 PRINT " or yourself"
8130 PRINT "then you will lose one of your
     five"
8140 PRINT "lives"
8145 PRINT
8150 PRINT "You get five screens of frogs to
     eat"
8160 PRINT "before you become a SUPER SNAKE!!"
8170 A$=INKEY$(2000)
8200 ENDPROC
```

Line 8060 and 8100 print examples of a frog and a toad for the player to see. The actual work of printing the word SNAKE is done by PROCsnake:

```
8500 DEF PROCsnake(X)
8510 DATA 2,2,2,1,0,2,2,2,2,0,2,2,2,2
8520 DATA 0,7,0,7,0,2,2,2,1
8530 DATA 8,0,0,0,0,8,0,0,10,0,8,0,0,10
8540 DATA 0,8,0,8,0,8,0,0,0
8550 DATA 5,5,5,5,0,8,0,0,10,0,8,3,1,10
8560 DATA 0,8,8,0,0,8,3,2,1
8570 DATA 0,0,0,8,0,8,0,0,10,0,8,0,0,10
8580 DATA 0,8,0,8,0,8,0,0,0
8590 DATA 3,2,2,2,0,3,0,0,9,0,3,0,0,9
8595 DATA 0,3,0,3,0,5,5,5,3
8600 RESTORE
8605 PRINT TAB(X,2)
8610 FOR I=1 TO 5
8620 PRINT TAB(X);
8630 FOR J=1 TO 23
8640 READ S
8650 IF S=0 THEN PRINT " "; ELSE
     PRINT CHR$(223+S);
8660 NEXT J
8670 PRINT
8680 NEXT I
8690 ENDPROC
```

The DATA statements in lines 8510 to 8595 are simply lists indicating characters to be printed. To reduce the number of digits that have to be typed in each DATA statement, 223 has to be added to each number to give the character code. For example, a 1 in the DATA statement will cause CHR$(224) to be printed. The exception to this rule is the use 0 to stand for blank. The DATA statements are organised in pairs and each pair defines a single row of 23 characters. Five rows of these standard size characters are used to build up the five-letter word SNAKE in large letters. There are other ways of creating large letters automatically but this is really the only convenient way of creating large letters using a range of different standard size characters.

PROCdraw_scene, PROCdraw_snake and PROCdraw_food

Although it isn't obvious at this stage in the program design,

PROCdraw_scene is in fact used by other procedures within the program to re-draw the scene when the snake has run into a poisonous toad or into itself. In this sense It Is not only part of the initialisation phase of the program, it is also part of the animation loop. The only modification that is necessary so that it can be used for both purposes is that it should print a number of frogs given by the value stored in FOOD, but if FOOD is zero it should first set it to a random value, thereby generating a new screenful of frogs.

```
2000 DEF PROCdraw_scene
2010 COLOUR 128+0
2020 CLS
2030 PROCdraw_snake
2050 COLOUR 3:PROCdraw_food(4*SC)
2060 IF FOOD=0 THEN FOOD=RND(10)+15
2070 COLOUR 2:PROCdraw_food(FOOD)
2080 COLOUR 1
2090 PRINT TAB(2,30);"Score=";STOTAL%+C%
2100 TIME=0
2110 REPEAT UNTIL TIME>=RND(100)+100
2120 ENDPROC
```

The call to PROCdraw_snake produces a fixed length snake at the samee starting position each time. PROCdraw_snake is also responsible for initialising the co-ordinate arrays X% and Y%, and the pointers Q%,md and Z%. The procedure PROCdraw_food will produce both frogs and toads depending on the setting of the foreground colour before it is called. Line 2050 uses it to print toads and line 2070 uses it to print frogs. The number of toads increases as each screenful of frogs is eaten, thus making the game progressively more difficult. The number of frogs is given by FOOD, which decreases from its initial value as frogs are eaten. If FOOD reaches zero line 2060 generates a random number of frogs to fill yet another screen. The final part of the procedure prints the total score and then waits for a random period of time before allowing the game to restart.

PROCdraw_snake looks a little complicated because, as mentioned above, it not only draws the snake, it also initialises the co-ordinate arrays X% and Y%.

```
2600 DEF PROCdraw_snake
2610 COLOUR 128+0:COLOUR 1
2620 X%=2:Y%=10
2625 XH%=X%:YH%=Y%
2630 H%=224
```

```
2640 S%=225
2650 T$=CHR$(226)
2660 PRINT TAB(X%,Y%);T$;
2670 X%(1)=X%:Y%(1)=Y%
2680 FOR Q%=2 TO 10
2690 X%=X%+1
2700 PRINT TAB(X%,Y%);CHR$(S%);
2710 X%(Q%)=X%:Y%(Q%)=Y%
2720 NEXT Q%
2730 X%=X%+1
2740 PRINT TAB(X%,Y%);CHR$(H%);
2750 X%(11)=X%:Y%(11)=Y%
2760 Z%=1:Q%=11
2770 XV%=1:YV%=0
2780 ENDPROC
```

The variables X%, Y% and XH%, YH% are used to record the current position of the snake's head. The reason for the use of two pairs of variables only becomes apparent when you look at PROCupdate, given later. H% and S% are used to hold the character codes of the current snake head and snake body characters and T$ holds the tail character. Line 2660 prints the tail character at 2,10 and then line 2670 stores its co-ordinates in X%(1) and Y%(1). The FOR loop, lines 2680 2720, prints the characters that make up the body of the snake and stores their co-ordinates in X%, and Y%. Finally, lines 2730 to 2750 print the head and then lines 2760 to 2770 initialise the variables that control the snake's length and initial direction of motion. The variable Z% points to the current tail co-ordinates in X% and Y%. and Q% points to the current head co-ordinates. The variables VX% and VY% control the direction of movement and are explained in more detail part of PROCget_direction and PROCupdate.

PROCdraw_food is very simple:

```
2500 DEF PROCdraw_food(AMOUNT)
2505 IF AMOUNT=0 THEN ENDPROC
2510 FOR K=1 TO AMOUNT
2520 U%=RND(40)-1
2530 V%=RND(26)
2540 IF FNC(U%,V%)<>0 THEN GOTO 2520
2550 PRINT TAB(U%,V%);CHR$(229);
2560 NEXT K
2570 ENDPROC
```

The FOR loop, lines 2510 to 2560, will print AMOUNT frogs or toads at random locations. Whether the procedure produces frogs or toads depends on what the foreground colour was set to before the procedure was called. Lines 2520 and 2530 generate random numbers to be used as the potential co-ordinates of a frog or toad. If there is already a character at U,V then line 2540 transfers control back so that another two random numbers are generated. The test for an existing character is perfomed by way of the function FNC which returns the colour of a pixel near the middle of the character location at U,V. (FNC was described in Chapter Two with reference to Ant Hill.)

```
9000 DEF FNC(X%,Y%)
9010 X%=16+32*X%
9020 Y%=1011-32*Y%
9030 =POINT(X%,Y%)
```

PROCone_move and its associated procedures

PROCone_move and simply calls two other procedures to do all of the necessary to move the snake:

```
3000 DEF PROCone_move
3010 PROCget_direction
3020 PROCupdate
3030 ENDPROC
```

PROCget_direction examines the keyboard to see which arrow key is pressed. If an arrow key is pressed then it updates the variables that indicate the direction of movement of the snake. The actual movement of the snake is produced by PROCupdate. This procedure not only updates the co-ordinate arrays and prints the necessary characters. it also checks to see if the snake has hit anything.

PROCget_direction is very similar to the other keyboard routines used in earlier programs:

```
4000 DEF PROCget_direction
4010 IF INKEY(-58) THEN XV%=0:YV%=-1:S%=231:
     H%=230
4020 IF INKEY(-42) THEN XV%=0:YV%=1:S%=233:
     H%=232
4030 IF INKEY(-26) THEN XV%=-1:YV%=0:S%=228:
     H%=227
4040 IF INKEY(-122) THEN XV%=1:YV%=0:S%=225:
     H%=224
```

```
4045 IF ADVAL(-5)<>15 THEN GOTO 4045
4046 SOUND 0,-15,6,1
4047 SOUND 0,0,6,1
4050 ENDPROC
```

The INKEY function is used to check each arrow key in turn (lines 4010 to 4040). If a particular arrow key is pressed then XV% and YV% are set to values that correspond to a velocity in that direction. For example, if the right arrow key is pressed XV is set to 1 and YV% is set to 0 and when VX% and VY% are added to X% and Y% (the current position of the head) they do indeed produce a movement to the right. As well as setting the velocity in response to the pressing of an arrow key, it is also necessary to select the appropriate character to be used for the snake's head and the snake's body. For example, if the right arrow key is pressed, line 4040 sets S% to a right pointing body character and H% to a right facing snake's head. The last part of the procedure makes a sort of hissing noise using channel 0 (lines 4045 to 4047).

PROCupdate is probably the most complicated procedure in the entire program:

```
4500 DEF PROCupdate
4510 XH%=X%:YH%=Y%
4520 X%=X%+XV%:Y%=Y%+YV%
4530 IF X%>39 THEN X%=0
4540 IF X%<0 THEN X%=39
4550 IF Y%>26 THEN Y%=0
4560 IF Y%<0 THEN Y%=26
4570 IF FNC(X%,Y%)<>0 THEN PROChit
4580 PRINT TAB(XH%,YH%);CHR$(S%);
4590 PRINT TAB(X%,Y%);CHR$(H%);
4600 Q%=Q%+1
4610 IF Q%>MAX% THEN Q%=1
4620 X%(Q%)=X%:Y%(Q%)=Y%
4630 PRINT TAB(X%(Z%),Y%(Z%));" ";
4640 Z%=Z%+1  4650 IF Z%>MAX% THEN Z%=1
4660 PRINT TAB(X%(Z%),Y%(Z%));T$;
4670 ENDPROC
```

Line 4510 saves the current position of the head, given by XY%,Y% in XH%,YH%. Then line 4520 updates the position of the head by add the velocities XV% and YV%. Following this, lines 4530 to 4560 sure that the new position Is still on the screen. Here are two ways of dealing with the situation where the snake is taken off the edge of the screen.

We could treat it as a mistake and make the snake lose a life or we could treat the screen as if its edges were joined together. That is. going off the top of the screen would make the snake reappear at the bottom edge. This is often known as a 'spherical universe' type of playing area and it is the strategy used in this game.

Line 4570 uses FNC to check if the snake has hit anything. If it has, then PROChit is called to deal with the situation. Otherwise line 4590 prints the head at its new position and line 4580 prints a body character at the head's old position so as to blank it out. Notice that PROCupdate doesn't have to worry about which of the many head or body characters to use because the correct ones have been selected by PROCCget_direction. Following this the new position of the head is stored in the co-ordinate arrays X% and Y%, (line 4620) after the head pointer Q%, has been updated (lines 4600 and 4610).

The final task of PROCupdate is to move the snake's tail. Line 4630 blanks out the tail at its old position. Lines 4640 and 4650 then update the tail pointer, Z%, and line 4660 prints the tail character at its new position

PROChit and its associated procedures

PROChit is responsible for producing the actions that are required wherever the snake runs into anything.

```
5000 DEF PROChit
5010 COL%=FNC(X%,Y%)
5020 IF COL%=3 OR COL%=1 THEN PROCtoad_self
5030 IF COL%=2 THEN PROCfrog
5040 ENDPROC
```

When PROChit is called all that is known is that the snake has run into something and the first thing that PROChit has to do is discover what! Line 5010 uses FNC to discover the colour of a pixel near the middle of the character location that the snake is about to move onto. If it is white or black then the snake has hit a toad or itself respectively and PROCtoad_self is called (line 3020) to take away one of the snake's lives. However, if the colour turns out to be green then PROCfrog is called to add one to the snake's score and make it one character longer (line 5030).

PROCtoad_self is quite a short procedure but the way that it works and interacts with the rest of the program is quite complicated.

```
5500 DEF PROCtoad_self
5505 SOUND 1,-15,10,20:SOUND 1,0,0,10
5506 IF ADVAL(-6)<>15 THEN GOTO 5506
5510 SOUND 1,-15,10,20:SOUND 1,0,0,10
5520 SN=SN+1
5530 IF SN>4 THEN STOTAL%=STOTAL%+C%:
     NO_LIVES=TRUE:ENDPROC
5540 PROCdraw_scene
5550 ENDPROC
```

Lines 5505 and 5506 make a noise to let the player know that a life Inn been lost. Then line 3520 adds one to the snake counter, SN. If SN h greater than 4 then all the lives have been used up and the game is over, Line 3530 tests for this and sets NO_LIVES to TRUE to inform the main program that this is the case. Otherwise PROCdraw_scene is called to re-draw the snake and the remaining frogs and toads ready lot the game to continue.

PROCfrog has two tasks to perform – looking after the score and making the snake one character longer.

```
5600 DEF PROCfrog
5610 SOUND 1,-15,250,1
5620 C%=C%+1:FOOD=FOOD-1
5630 PRINT TAB(2,30);"Score=";STOTAL%+C%
5640 Z%=Z%-1
5650 IF Z%<1 THEN Z%=MAX%
5660 IF FOOD<>0 THEN ENDPROC
5670 STOTAL%=STOTAL%+C%
5680 C%=0
5690 IF SC=4 THEN NO_FOOD=TRUE:ENDPROC
5700 SC=SC+1
5710 PROCdraw_scene
5720 ENDPROC
```

Line 5610 makes a noise to indicate to the player that a frog has just been eaten. Then line 5620 adds one to the current score and subtract one from FOOD. The reason that FOOD has to be reduced by one that it is used to test Whether all the frogs have been eaten and as a indicator of how many frogs should be redrawn if the snake loses a life ad the game is restarted. Line 5630 prints the total score. Increasing the length of the snake by one character is simplicity itself. Lines 5640 and 5650 adjust the tail pointer Z%, moving the current location oft tail back by one position. When PROCfrog ends, control is returned to PROCupdate which is in the process of moving the snake forward. Because of this the effect of adjusting Z% is to leave the tail in its

current location for one move even though the head moves on. In this way the length of the snake is increased automatically as it moves along.

The final part of PROCfrog is concerned with detecting whether all the frogs have been eaten. If this is the case, line 5670 adds C% to the total. Line 5690 then tests to see if all five screenfuls of frogs have been eaten and sets NO_FOOD to TRUE if this is the case. If there are more screenfuls of frogs still to be eaten then the scene number SC is incremented and PROCdraw screen is called to draw a new screenful of frogs. Notice that PROCdraw_scene will detect the fact that FOOD is zero and so generate a new random number of frogs.

PROCend_game

The end of the snake game is fairly unspectacular in that it simply prints an appropriate message telling you how well you have done, based on the number of screenfuls of frogs you have eaten before losing all five lives.

```
7000 DEF PROCend_game
7010 CLS
7020 COLOUR 1
7030 IF SC=0 THEN PRINT TAB(3,10);"Demoted to
     slowworm"
7040 IF SC=1 THEN PRINT TAB(3,10);"The frogs
     don't have to worry"
7050 IF SC=2 THEN PRINT TAB(3,10);"Not bad for
     a grass snake"
7060 IF SC=3 THEN PRINT TAB(3,10);"Well
     slithered"
7070 IF SC=4 THEN PRINT TAB(3,10);"A venomous
     performance"
7074 IF SC=4 AND NO_FOOD THEN PRINT TAB(5);
     "you're a SUPER SNAKE"
7075 *FX 15,0
7080 FOR I=5 TO 1 STEP -1
7090 IF STOTAL%>=T(I) THEN N=1
7100 NEXT I
7105 PRINT
7110 IF N=0 THEN PRINT TAB(10);"You scored ";
     STOTAL%:GOTO 7500
7120 PRINT TAB(10);"You are now ranked ";N
7130 PRINT TAB(10);"in the snake league"
7140 IF N=5 THEN GOTO 7200
7150 FOR I=5 TO N+1 STEP -1
```

```
7160 T(I)=T(I-1)
7170 T$(I)=T$(I-1)
7180 NEXT I
7200 T(N)=STOTAL%
7205 PRINT TAB(10);
7210 INPUT "What is your name",T$(N)
7220 CLS
7230 FOR I=1 TO 5
7240 PRINT TAB(5,I*2+5);T$(I);TAB(20);T(I)
7250 NEXT I
7500 PRINT
7510 INPUT "Another slither Y/N",A$
7520 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 7510
7530 IF LEFT$(A$,1)="N" THEN AGAIN=FALSE ELSE
     AGAIN=TRUE
7540 ENDPROC
```

Lines 7030 to 7074 print the congratulatory messages. An additional feature of the end of the snake game is that it maintains a league table of scores. If you are in the top five scores then it asks for your name and inserts it into the table at the correct position. Lines 7080 to 7100 find out where the score fits in the current league table. This works by finding the first score that your score is greater than or equal to in the league table. If you deserve to be placed in the league table all the scores lower than or equal to yours are moved down one place (lines 7150 to 7180) and your score and name are inserted (lines 7200 to 7210). Then the league table is printed (lines 7220 to 7250) and the player is asked if another game is requested (lines 7500 to 7530). Notice that if another game is required it isn't good enough to simply use RUN to restart it as this would erase the contents of the arrays that hold the league table! Instead, the variable AGAIN is used to indicate to the rest of the program that it is being re-run.

Conclusion - playing the game

Even this simple game involving a snake is quite addictive! This is not to say that there's no scope both for improvement and innovation, but overall the game is a great success and has a speed and interest normally only found in assembly language games. In the next chapter the potential using the snake in another game will be explored.

The final version - a complete listing

```
   10 REM SNAKE
   20 MODE 1
   30 AGAIN=FALSE
   40 PROCinitialise
   50 PROCtitle
   60 PROCdraw_scene
   70 REPEAT
   80 PROCone_move
   90 UNTIL NO_FOOD OR NO_LIVES
  100 PROCend_game
  110 IF AGAIN THEN GOTO 40
  120 END

 1000 DEF PROCinitialise
 1010 VDU 23,224,&38,&7C,&FA,&FF,&FF,&FA,&7C,&38
 1020 VDU 23,225,&7E,&EF,&F7,&FB,&FB,&F7,&EF,&7E
 1030 VDU 23,226,&18,&3C,&7E,&FF,&FF,&7E,&3C,&18
 1040 VDU 23,227,&1C,&3E,&5F,&FF,&FF,&5F,&3F,&1C
 1050 VDU 23,228,&7E,&F7,&EF,&DF,&DF,&EF,&F7,&7E
 1060 VDU 23,229,&82,&54,&38,&DA,&FE,&82,&44,&82
 1070 VDU 23,230,&18,&3C,&5A,&FF,&FF,&FF,&7E,&7E
 1080 VDU 23,231,&7E,&FF,&E7,&D8,&BD,&FF,&FF,&7E
 1090 VDU 23,232,&3C,&7E,&FF,&FF,&FF,&5A,&3C,&18
 1100 VDU 23,233,&7E,&FF,&FF,&BD,&DB,&E7,&FF,&7E
 1110 VDU 19,0,3,0,0,0:REM 0=YELLOW
 1120 VDU 19,1,0,0,0,0:REM 1=BLACK
 1130 VDU 19,2,2,0,0,0:REM 2=GREEN
 1140 VDU 19,3,7,0,0,0:REM 3=WHITE
 1150 IF AGAIN THEN GOTO 1260
 1200 DIM X%(50),Y%(50)
 1205 MAX%=50
 1210 DIM T$(5)
 1220 DIM T(5)
 1230 FOR I=1 TO 5
 1240 T(I)=0
 1250 NEXT I
 1260 NO_FOOD=FALSE
 1270 NO_LIVES=FALSE
 1280 HIT=FALSE
 1290 C%=0:STOTAL%=0
 1300 SC=0:SN=0:FOOD=0
 1310 ENDPROC
```

```
2000 DEF PROCdraw_scene
2010 COLOUR 128+0
2020 CLS
2030 PROCdraw_snake
2050 COLOUR 3:PROCdraw_food(4*SC)
2060 IF FOOD=0 THEN FOOD=RND(10)+15
2070 COLOUR 2:PROCdraw_food(FOOD)
2080 COLOUR 1
2090 PRINT TAB(2,30);"Score=";STOTAL%+C%
2100 TIME=0
2110 REPEAT UNTIL TIME>=RND(100)+100
2120 ENDPROC


2500 DEF PROCdraw_food(AMOUNT)
2505 IF AMOUNT=0 THEN ENDPROC
2510 FOR K=1 TO AMOUNT
2520 U%=RND(40)-1
2530 V%=RND(26)
2540 IF FNC(U%,V%)<>0 THEN GOTO 2520
2550 PRINT TAB(U%,V%);CHR$(229);
2560 NEXT K
2570 ENDPROC


2600 DEF PROCdraw_snake
2610 COLOUR 128+0:COLOUR 1
2620 X%=2:Y%=10
2625 XH%=X%:YH%=Y%
2630 H%=224
2640 S%=225
2650 T$=CHR$(226)
2660 PRINT TAB(X%,Y%);T$;
2670 X%(1)=X%:Y%(1)=Y%
2680 FOR Q%=2 TO 10
2690 X%=X%+1
2700 PRINT TAB(X%,Y%);CHR$(S%);
2710 X%(Q%)=X%:Y%(Q%)=Y%
2720 NEXT Q%
2730 X%=X%+1
```

```
2740 PRINT TAB(X%,Y%);CHR$(H%);
2750 X%(11)=X%:Y%(11)=Y%
2760 Z%=1:Q%=11
2770 XV%=1:YV%=0
2780 ENDPROC


3000 DEF PROCone_move
3010 PROCget_direction
3020 PROCupdate
3030 ENDPROC


4000 DEF PROCget_direction
4010 IF INKEY(-58) THEN XV%=0:YV%=-1:S%=231:
     H%=230
4020 IF INKEY(-42) THEN XV%=0:YV%=1:S%=233:
     H%=232
4030 IF INKEY(-26) THEN XV%=-1:YV%=0:S%=228:
     H%=227
4040 IF INKEY(-122) THEN XV%=1:YV%=0:S%=225:
     H%=224
4045 IF ADVAL(-5)<>15 THEN GOTO 4045
4046 SOUND 0,-15,6,1
4047 SOUND 0,0,6,1
4050 ENDPROC


4500 DEF PROCupdate
4510 XH%=X%:YH%=Y%
4520 X%=X%+XV%:Y%=Y%+YV%
4530 IF X%>39 THEN X%=0
4540 IF X%<0 THEN X%=39
4550 IF Y%>26 THEN Y%=0
4560 IF Y%<0 THEN Y%=26
4570 IF FNC(X%,Y%)<>0 THEN PROChit
4580 PRINT TAB(XH%,YH%);CHR$(S%);
4590 PRINT TAB(X%,Y%);CHR$(H%);
4600 Q%=Q%+1
4610 IF Q%>MAX% THEN Q%=1
4620 X%(Q%)=X%:Y%(Q%)=Y%
4630 PRINT TAB(X%(Z%),Y%(Z%));" ";
4640 Z%=Z%+1
4650 IF Z%>MAX% THEN Z%=1
4660 PRINT TAB(X%(Z%),Y%(Z%));T$;
4670 ENDPROC
```

```
 5000 DEF PROChit
 5010 COL%=FNC(X%,Y%)
 5020 IF COL%=3 OR COL%=1 THEN PROCtoad_self
 5030 IF COL%=2 THEN PROCfrog
 5040 ENDPROC


 5500 DEF PROCtoad_self
 5505 SOUND 1,-15,10,20:SOUND 1,0,0,10
 5506 IF ADVAL(-6)<>15 THEN GOTO 5506
 5510 SOUND 1,-15,10,20:SOUND 1,0,0,10
 5520 SN=SN+1
 5530 IF SN>4 THEN STOTAL%=STOTAL%+C%:
      NO_LIVES=TRUE:ENDPROC
 5540 PROCdraw_scene
 5550 ENDPROC


 5600 DEF PROCfrog
 5610 SOUND 1,-15,250,1
 5620 C%=C%+1:FOOD=FOOD-1
 5630 PRINT TAB(2,30);"Score=";STOTAL%+C%
 5640 Z%=Z%-1
 5650 IF Z%<1 THEN Z%=MAX%
 5660 IF FOOD<>0 THEN ENDPROC
 5670 STOTAL%=STOTAL%+C%
 5680 C%=0
 5690 IF SC=4 THEN NO_FOOD=TRUE:ENDPROC
 5700 SC=SC+1
 5710 PROCdraw_scene
 5720 ENDPROC


 7000 DEF PROCend_game
 7010 CLS
 7020 COLOUR 1
 7030 IF SC=0 THEN PRINT TAB(3,10);"Demoted to
      slowworm"
 7040 IF SC=1 THEN PRINT TAB(3,10);"The frogs
      don't have to worry"
 7050 IF SC=2 THEN PRINT TAB(3,10);"Not bad for
      a grass snake"
 7060 IF SC=3 THEN PRINT TAB(3,10);"Well
slithered"
 7070 IF SC=4 THEN PRINT TAB(3,10);"A venomous
      performance"
 7074 IF SC=4 AND NO_FOOD THEN PRINT
TAB(5);"you're a SUPER SNAKE"
 7075 *FX 15,0
```

```
7080 FOR I=5 TO 1 STEP -1
7090 IF STOTAL%>=T(I) THEN N=1
7100 NEXT I
7105 PRINT
7110 IF N=0 THEN PRINT TAB(10);"You scored ";
     STOTAL%:GOTO 7500
7120 PRINT TAB(10);"You are now ranked ";N
7130 PRINT TAB(10);"in the snake league"
7140 IF N=5 THEN GOTO 7200
7150 FOR I=5 TO N+1 STEP -1
7160 T(I)=T(I-1)
7170 T$(I)=T$(I-1)
7180 NEXT I
7200 T(N)=STOTAL%
7205 PRINT TAB(10);
7210 INPUT "What is your name",T$(N)
7220 CLS
7230 FOR I=1 TO 5
7240 PRINT TAB(5,I*2+5);T$(I);TAB(20);T(I)
7250 NEXT I
7500 PRINT
7510 INPUT "Another slither Y/N",A$
7520 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 7510
7530 IF LEFT$(A$,1)="N" THEN AGAIN=FALSE ELSE
     AGAIN=TRUE
7540 ENDPROC


8000 DEF PROCtitle
8005 COLOUR 128+1:COLOUR 0
8006 CLS
8007 VDU 23,1,0;0;0;0;
8010 PROCsnake(5)
8020 PRINT TAB(2,10);
8030 PRINT "You must guide your snake using the"
8040 PRINT "four arrow keys and eat all the
     green"
8050 PRINT "frogs ";
8060 COLOUR 2:PRINT CHR$(229)
8070 COLOUR 0
8080 PRINT
8090 PRINT " If you try to eat a toad ";
8100 COLOUR 3:PRINT CHR$(229);
8110 COLOUR 0
8120 PRINT " or yourself"
```

```
8130 PRINT "then you will lose one of your five"
8140 PRINT "lives"
8145 PRINT
8150 PRINT "You get five screens of frogs to
eat"
8160 PRINT "before you become a SUPER SNAKE!!"
8170 A$=INKEY$(2000)
8200 ENDPROC

8500 DEF PROCsnake(X)
8510 DATA 2,2,2,1,0,2,2,2,2,0,2,2,2,2
8520 DATA 0,7,0,7,0,2,2,2,1
8530 DATA 8,0,0,0,0,8,0,0,10,0,8,0,0,10
8540 DATA 0,8,0,8,0,8,0,0,0
8550 DATA 5,5,5,5,0,8,0,0,10,0,8,3,1,10
8560 DATA 0,8,8,0,0,8,3,2,1
8570 DATA 0,0,0,8,0,8,0,0,10,0,8,0,0,10
8580 DATA 0,8,0,8,0,8,0,0,0
8590 DATA 3,2,2,2,0,3,0,0,9,0,3,0,0,9
8595 DATA 0,3,0,3,0,5,5,5,3
8600 RESTORE
8605 PRINT TAB(X,2)
8610 FOR I=1 TO 5
8620 PRINT TAB(X);
8630 FOR J=1 TO 23
8640 READ S
8650 IF S=0 THEN PRINT " "; ELSE
     PRINT CHR$(223+S);
8660 NEXT J
8670 PRINT
8680 NEXT I
8690 ENDPROC

9000 DEF FNC(X%,Y%)
9010 X%=16+32*X%
9020 Y%=1011-32*Y%
9030 =POINT(X%,Y%)
```

Chapter Six
Tadpole

The animated snake developed in the last chapter can be used to create a wide variety of exciting games. Tadpole is just such a game and this chapter shows how the Snake program can be modified to produce a new game. In the second half of the chapter we take the opportunity to change one of the complicated procedures within the animation loop machine code. This results in a much faster and more exciting game and serves to illustrate the way that BASIC and machine code should be used together.

The game design

The basic idea behind Tadpole is similar to that developed in Snake in that an animated snake has to eat frogs to score points. The difference is that in Tadpole the frogs move! At the start of the game the frogs are all still tadpoles swimming in a pond in the middle of the screen. One by we they turn into frogs and make a rush for the edge of the screen. The snake cannot eat the tadpoles until they have become frogs, and indeed it loses a life if it falls into the pond. The object of the game is simply to eat as many frogs as possible before they make their escape off the edge of the screen.

Apart from the addition of the pond and the moving frogs the rest of the elements of tadpole remain unchanged from the basic Snake game. In particular, the snake has five fives and five screenfuls of frogs to eat up, If the snake loses a life then the remaining frogs miraculously turn back into tadpoles and the snake has another opportunity to lie in wait for. them. All the graphics characters used in Tadpole have already been introduced as part of the Snake program and, apart from the pond, the screen layout (Fig. 6. I) is very similar to that produced by Snake but it should be recalled that in Snake the frogs are static, whereas in Tadpole they move.

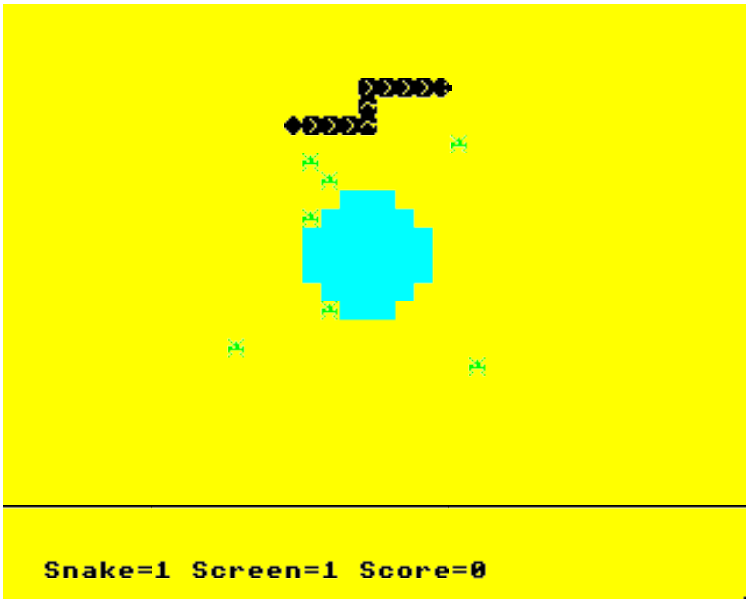


Fig. 6.1.

Setting the scene

The modifications to Snake fall into two parts, firstly the changes and additions necessary to set up the game and secondly the changes and additions necessary to animate the frogs. This section deals with the setting up of the game.

Most of the modifications necessary to set up the game are made to one procedure PROCdraw_scene. Now, instead of printing frogs and toads, it has to print a pond containing tadpoles in the middle of the screen. The pond is best produced by way of a new procedure, PROCdraw_pool, and the tadpoles by a modification to PROCdraw_food. The new version of PROCdraw_scene is:

```
2000 DEF PROCdraw_scene
2010 COLOUR 128+0
2020 CLS
2030 PROCdraw_snake
2040 PROCdraw_pool(18,10)
2045 COLOUR 128+3
2060 IF FOOD=0 THEN FOOD=NUM%
```

```
2070 COLOUR 1:PROCdraw_food(FOOD)
2080 COLOUR 128+0
2090 PRINT TAB(2,30);"Snake=";SN+1;" Screen=";
     SC+1;" Score=";STOTAL%+C%
2095 GCOL 0,1
2096 MOVE 0,156
2097 DRAW 1280,156
2100 TIME=0
2110 REPEAT UNTIL TIME>=RND(100)+100
2120 ENDPROC
```

The new procedure to draw the pond is called at line 2040. PROCCdraw_food is now only called once (line 2070) to print a fixed number of tadpoles in the pond. The only other changes are the addition of line 2090 to print the snake and screen number and the inclusion of high resolution graphics commands, lines 2095 to 2097, to draw a horizontal line to show the limit of the playing area.

PROCdraw_pool is nothing more than a list of PRINT statements that print spaces using a blue background colour:

```
2200 DEF PROCdraw_pool(X,Y)
2210 COLOUR 128+3
2220 PRINT TAB(X,Y);SPC(3);
2230 PRINT TAB(X-1,Y+1);SPC(5);
2240 PRINT TAB(X-2,Y+2);SPC(7);
2250 PRINT TAB(X-2,Y+3);SPC(7);
2260 PRINT TAB(X-2,Y+4);SPC(7);
2270 PRINT TAB(X-1,Y+5);SPC(5);
2280 PRINT TAB(X,Y+6);SPC(3);
2290 ENDPROC
```

The need to use a blue background colour implies that one of the logical to physical colour assignments in PROCinitiaiise has to be changed. As there are no white toads in Tadpole the logical colour selected to be changed is 3:

```
1140 VDU 19,3,6,0,0,0:REM 3=CYAN
```

Cyan is used rather than blue because it produces a better contrast with the other colours on a black and white set.

PROCdraw_food now has to place tadpoles in the pond rather than frogs anywhere on the screen. This is easily catered for by restricting the range of the random numbers generated by lines 2520 and 2530. Later in the program the tadpoles and the frogs that they give rise have to be animated and this implies that their positions should be known. To

keep track of where all the tadpoles and frogs are it is necessary to use a pair of arrays to hold the x and y co-ordinates of each tadpole/frog.

```
2500 DEF PROCdraw_food(AMOUNT)
2505 IF AMOUNT=0 THEN ENDPROC
2510 FOR K=1 TO AMOUNT
2520 U%=RND(7)+15
2530 V%=RND(7)+9
2540 IF FNC(U%,V%)<>3 THEN GOTO 2520
2550 PRINT TAB(U%,V%);",";
2555 FX%(K)=U%:FY%(K)=V%
2560 NEXT K
2570 IF AMOUNT=NUM% THEN ENDPROC
2580 FOR K=AMOUNT+1 TO NUM%
2585 NEXT K
2590 ENDPROC
```

Line 2555 saves the co-ordinates in FX% and FY% for later use. The maximum number of tadpoles/frogs is eight but at later stages of the game there may be fewer due to frogs escaping or being eaten. To indicate that a tadpole? frog is not being used in the game its x coordinate is set to −1. The FOR loop, lines 2580 to 2586, sets the unused entries of the array FX% to −1. Of course, the new pair of arrays has to be dimensioned and so lines 1206 and 1207 have to be added to PROCinit:

```
1206 NUM%=8

1207 DIM FX%(NUM%),FY%(NUM%)
```

Making an escape

Once the initial display has been set up along with the co-ordinates of all the tadpoles in the pair of arrays FX% and FY%, the animation of the snake and the frogs can begin. The only change to the animation loop is the addition of a call to PROCmove_f which moves a single frog at random. That is, PROCone_move becomes

```
3000 DEF PROCone_move
3005 PROCmove_f
3010 PROCget_direction
```

```
3020 PROCupdate
3030 ENDPROC
```

PROCmove_f is a little more complicated than you might expect. The main problems lie in making sure that the frog doesn't land on the snake, using the correct background colour to print and blank the frog and detecting when a frog has completed its escape. In fact, most of the solutions to these problems have been introduced in earlier chapters.

```
3500 DEF PROCmove_f
3510 K%=RND(NUM%)
3520 IF FX%(K%)=-1 THEN ENDPROC
3530 N%=FNC(FX%(K%),FY%(K%))
3535 IF N%=3 THEN COLOUR 128+3
3536 IF N%<>1 THEN PRINT
     TAB(FX%(K%),FY%(K%));" ";
3540 FX%(K%)=FX%(K%)-SGN(19.5-FX%(K%))
3550 FY%(K%)=FY%(K%)-SGN(13.5-FY%(K%))
3560 IF FX%(K%)<0 OR FX%(K%)>39 THEN
     PROCfrog_gone:ENDPROC
3570 IF FY%(K%)<0 OR FY%(K%)>26 THEN
     PROCfrog_gone:ENDPROC
3574 COLOUR 128
3575 N%=FNC(FX%(K%),FY%(K%))
3576 IF N%<>0 THEN GOTO 3540
3580 COLOUR 2
3590 PRINT TAB(FX%(K%),FY%(K%));CHR$(229);
3595 COLOUR 1
3600 ENDPROC
```

Line 3510 selects a tadpole/frog to move at random. If the x co-ordinate of this tadpole/frog is −1 then it has either been eaten or it has already escaped and so fine 3520 returns control to PROCone_move. Line 3530 finds the colour of the background of the frog's old position and stores it in N%. As long as this colour isn't black then line 3536 blanks out the old position. Lines 3540 and 3550 update the frog's co-ordinates in such a way that the frog moves toward the edge of the screen. The way that this works is easy to understand once you know that SGN(19.5–FX%(K%)) is +1 if the frog is to the left of the centre of the pond and –1 if it is to the right of the centre of the pond. Thus subtracting SGN(19.5–FX%(K%))) from the current x co-ordinate of the frog always results in it moving away from the centre of the pond. A similar argument shows that subtracting SGN(13.5–FY%(K%)) from the current y co-ordinate of the frog also results in it moving away from the centre of the pond. Following tins lines 3560 and 3570 test for the frog

going off the screen and call PROCfrog gone if it has. The final part of the procedure, lines 3574 to 3595, prints the frog at its new position using the correct background colour. Once again the background colour is determined by a call to FNC (line 3575).

The PROCmove_f calls PROCfrog_gone to make the fact that a frog has escaped:

```
3900 DEF PROCfrog_gone
3910 FX%(K%)=-1
3920 FOOD=FOOD-1
3930 IF FOOD<>0 THEN ENDPROC
3935 STOTAL%=STOTAL%+C%
3936 C%=0
3940 IF SC=4 THEN NO_FOOD=TRUE:ENDPROC
3950 SC=SC+1
3960 PROCdraw_scene
3970 ENDPROC
```

Line 3910 sets the frog's horizontal co-ordinate to –1 to indicate to the rest of the program that it has escaped. The rest of the procedure subtracts one from FOOD and restarts the game with a new screenful of tadpoles/frogs unless of course five screenfuls have already escaped or been eaten.

Catching frogs

The remaining changes to the program are mainly to PROCfrog, PROCfrog is called whenever the snake runs into a green object, i.e. a frog! In the original version of Snake, all PROCfrog had to do was add one to the score and check for the possibility that all the frogs had been eaten. Now it also has to remove the frog from the list of co-ordinates in FX%$ and FY9b. This is done by adding a call to a new procedure, PROCrem_f:

```
5655 PROCrem_f
```

In addition, the opportunity is taken to alter line 5630 to print the snake and screen numbers:

```
5630 PRINT TAB(2,30);"Snake=";SN+1;" Screen=";
     SC+1;" Score=";STOTAL%+C%
```

The position of the frog the the snake has just run into is stored in

X% and Y%. PROCrem_f removes it from the list of frogs by searching FX% and FY% for a frog with the same co-ordinates and setting its x co-ordinate to –1

```
5800 DEF PROCrem_f
5810 K%=0
5820 REPEAT
5830 K%=K%+1
5840 UNTIL X%=FX%(K%) AND Y%=FY%(K%)
5850 FX%(K%)=-1  5860 ENDPROC
```

The finishing touches to Tadpole involve altering the messages issued by PROCend_game:

```
7030 IF STOTAL%<10 THEN PRINT TAB(3,10);
     "Demoted to slowworm":GOTO 7075
7040 IF STOTAL%<20 THEN PRINT TAB(3,10);
     "The frogs don't have to worry":GOTO 7075
7050 IF STOTAL%<30 THEN PRINT TAB(3,10);
     "Not bad for a grass snake":GOTO 7075
7060 IF STOTAL%<35 THEN PRINT TAB(3,10);
     "Well slithered":GOTO 7075
7070 PRINT TAB(3,10);"A venomous performance"
7074 PRINT TAB(5);"you're a SUPER SNAKE"
```

and the instructions in PROCtitle:

```
8080 PRINT
8090 PRINT " Don't fall in the water or try to"
8120 PRINT "eat yourself or you will lose
     one of"
8130 PRINT "your five lives"
8140 PRINT
8145 PRINT
```

Adding assembler

If you make all the modifications listed above to the Snake program given in the last chapter then you will indeed have a new game – Tadpole. However, rather than leave the program at this stage of development PROCmove_f will he changed into a machine code routine, partly to increase the speed of the program but also to show the

general principles involved in adding assembly language to an existing program.

The actual translation of PROCmove_f to assembler is not as difficult as you would expect; the real difficulty stems from the extreme shortage of memory. In fact memory is so short that the final version of the program has to be loaded into memory in two stages the BASIC part of the program and the machine code part of the program. However, trying to develop a program of this size in two pieces is very difficult and during the development of the machine code routine the need to run the program as a single unit was so great that extra fr, memory was acquired by the simple expedient of deleting every section of the program that wasn't absolutely essential. In other words, PROCtitle and PROCend_game were both dispensed with temporarily. Once the machine code was fully debugged the program was put back together again to produce a master copy from which the machine code section and the BASIC section can be produced. It is this master copy that is listed at the end of this chapter and it is important to realise that this program is much too large to fit into the memory remaining after Mode 1 has taken its 20K. How to convert into two modules that fit into the remaining space is described in this chapte's final section.

An assembly language version of PROCmove_f

To convert PROCmove_f to machine code it is necessary to add a whole new procedure that uses the 6502 assembler to produce machine code from assembly language. This new procedure is only called oats at the start of the program and from then on the machine code that [I produces is available for use. Normally machine code would be stored in a byte array but in this case memory is in such short supply that it will be stored in the area normally reserved for use as serial data buffers, that is, &900 to &AFF. The call to PROCasmb is added to PROCinitialise at line 1208:

```
1208 PROCasmb
```

As well as this change it is also necessary to reduce the dimension oft arrays X% and Y% to free some additional memory and delete the lines that define the arrays FX% and FY% as these will be set up within the machine code. To achieve this. change lines 1200 and 1205

```
1200 DIM X%(15),Y%(15)
1205 MAX%=15
```

and delete line 1206.

The best way to deal with the changes to PROCmove_f itself is to delete the existing version and type in:

```
 3500 DEF PROCmove_f
 3510 A%=RND(NUM%)
 3520 B%=USR(move_f%)
 3530 IF ?FLAG%<>0 AND FX%?A%<>&FF THEN
      PROCfrog_gone
 3540 ENDPROC
```

Now the procedure uses the machine code routine move_f to do most of the work. The number of the frog to be moved is passed to move_f in the resident integer variable A%. On return from move_f the memory location FLAG% is 0 if a frog was moved and equal to &FF otherwise. This is used by line 3530 to test for frogs that have escaped. Notice that now the arrays FX% and FY% have been replaced by byte arrays (defined within the machine code). The x co-ordinate of the frog whose number is in A% is given by FX%?A% and similarly its y co-ordinate is in FY%?A%. Other sections of the program that used the arrays FX% and FY% also have to be changed. PROCfrog_gone becomes:

```
 3900 DEF PROCfrog_gone
 3910 FX%?A%=&FF
 3920 FOOD=FOOD-1
 3930 IF FOOD<>0 THEN ENDPROC
 3935 STOTAL%=STOTAL%+C%
 3936 C%=0
 3940 IF SC=4 THEN NO_FOOD=TRUE:ENDPROC
 3950 SC=SC+1
 3960 PROCdraw_scene
 3970 ENDPROC
```

and PROCrem_f becomes:

```
 5800 DEF PROCrem_f
 5810 K%=0
 5820 REPEAT
 5830 K%=K%+1
 5840 UNTIL X%=FX%?K% AND Y%=FY%?K%
 5850 FX%?K%=&FF
 5860 ENDPROC
```

Notice that &FF is now used to mark the fact that a frog is not in use rather than –1.

These are all the changes that are necessary to the program apart from the addition of PROCasmb. To explain the exact workings of the assembly language would take far too long, so instead we have adopted the alternative strategy of presenting a heavily commented listing. As the assembly language does exactly the same job as the original BASIC version of PROCmove_f it should be easy to understand. However to make sure that you do not run out of memory do not type in the assembly language comments and do not insert any unnecessary spaces.

```
6000 DEF PROCasmb
6004 CODE%=&900
6005 OSWORD%=&FFF1
6010 OSWRCH%=&FFEE
6016 ATEMP%=&70
6017 FLAG%=&71
6018 PARM%=&72:COL%=PARM%+4
6019 FX%=&77:FY%=&80
6020 FOR PASS=0 TO 3 STEP 3
6030 P%=CODE%
6040 [OPT PASS
6050 .move_f% STA ATEMP%  \A contains the
                           'frog number'
6060 JSR  get_cords%      \get frog's position
                           in X and Y
6070 CPX  #&FF            \is the frog in use
6080 BNE  skip1%
6090 STX  FLAG%           \if not in use set
                           flag to &FF
6100 RTS                  \and return
6110 .skip1% JSR f_col%   \find the colour
                           at X,Y
6120 CMP  #3  \is it 3?
6130 BNE  skip2%

6140 LDA  #128+3          \if it is then set
                           background
```

```
6150 JSR  s_col%          \colour to 3
6160 .skip2%  LDA COL%    \get the colour
                           again
6170 CMP  #1              \is it 1?
6180 BEQ  skip3%
6190 JSR  tab%            \if it isn't then
                          TAB(X,Y)
6200 LDA  #32             \and print a blank
6210 JSR  OSWRCH%
6220 .skip3% JSR update%  \update X and Y
6230 JSR  gone%           \has the frog
                           escaped?
6240 BCC  skip4%          \carry set if it has
6250 STA  FLAG%           \set FLAG to &FF
6260 RTS                  \and return
6270 .skip4%  LDA #128    \set background
                           to colour
6280 JSR  s_col%          \zero
6290 JSR  f_col%          \find colour at X,Y
6300 BNE  skip3%          \if it is not zero
                           then update again
6310 LDA  #2              \set foreground
                           colour to 2
6320 JSR  s_col%
6330 JSR  tab%            \TAB(X,Y)
6340 LDA  #229            \print a frog
6350 JSR  OSWRCH%
6360 LDA  #1              \set foreground
6370 JSR  s_col%          \colour to 1
6380 JSR  s_cords%        \store X and Y in
                           FX%?A% and FY%?A%
6390 LDA  #0
6400 STA  FLAG%           \set FLAG to zero
6410 RTS  \and return
6420 \

6430 .get_cords% LDX ATEMP% \gets co-ordinates
6440 LDY  FY%,X           \from FY%?A% into Y
6450 LDA  FX%,X           \and FX%?A% into X
6460 TAX
6470 RTS
6480 \

6490 .f_col%  STX PARM%   \equivalent to FNC
6500 LDA  #0
6510 STA  PARM%+1         \X*32
6520 ASL  PARM%:ROL PARM%+1
```

```
6521 ASL  PARM%:ROL PARM%+1
6522 ASL  PARM%:ROL PARM%+1
6523 ASL  PARM%:ROL PARM%+1
6524 ASL  PARM%:ROL PARM%+1
6530 LDA  #16 \+16
6540 CLC
6550 ADC  PARM%
6560 STA  PARM%
6565 LDA  #0
6566 ADC  PARM%+1
6567 STA  PARM%+1
6570 STY  PARM%+2
6590 LDA  #0
6600 STA  PARM%+3
6610 ASL  PARM%+2:ROL PARM%+3 \Y*32
6611 ASL  PARM%+2:ROL PARM%+3
6612 ASL  PARM%+2:ROL PARM%+3
6613 ASL  PARM%+2:ROL PARM%+3
6614 ASL  PARM%+2:ROL PARM%+3
6620 LDA  #&F3           \1011-
6624 SEC
6625 SBC  PARM%+2
6630 STA  PARM%+2
6631 LDA  #&3
6632 SBC  PARM%+3
6633 STA  PARM%+3
6650 TXA  \save X
6660 PHA
6670 TYA  \and Y
6680 PHA
6690 LDX  #(PARM% MOD 256)
6700 LDY  #(PARM% DIV 256)
6710 LDA  #9
6720 JSR  OSWORD%
6730 PLA  \restore X
6740 TAY
6750 PLA  \and Y
6760 TAX
6770 LDA  PARM%+4        \get result in A
6780 RTS
6790 \
6800 .s_col%  PHA        \sets colour to A
6801 LDA  #17
6802 JSR  OSWRCH%
6803 PLA
```

```
6804 JSR  OSWRCH%
6805 RTS
6806 \
6807 .tab%    LDA #31     \equivalent to
                          TAB(X,Y)
6808 JSR  OSWRCH%
6809 TXA
6810 JSR  OSWRCH%
6811 TYA
6812 JSR  OSWRCH%
6813 RTS
6814 \

6820 .update% CPX #19     \equivalent to
6825 BPL  u1%             \X=X-SGN(19-X)
6830 DEX
6835 DEX
6840 .u1% INX
6845 CPY  #13 \Y=Y-SGN(13-Y)
6850 BPL  u2%
6855 DEY
6860 DEY
6865 .u2% INY
6870 RTS
6875 \

6880 .gone%   CPX #0      \tests to see if
6882 BMI  g1%             \frog has escaped
6884 CPX  #39             \carry flag clear if
6886 BPL  g1%             \frog still on screen
6888 CPY  #0
6890 BMI  g1%
6892 CPY  #26
6893 BPL  g1%
6894 CLC
6895 RTS
6896 .g1% LDA #&FF
6897 SEC
6898 RTS
6899 \

6900 .s_cords%  TXA       \store X and Y in
6901 LDX  ATEMP%          \FX%?A% and FY%?A%
6902 STA  FX%,X
6903 TYA
6904 STA  FY%,X
6905 RTS
6910 ]
6920 NEXT PASS
6930 ENDPROC
```

The X and Y referred to in the above comments are of course the X and Y registers which are used throughout the language routine to hold the

x and y co-ordinates of the frog. Notice the way that subroutines are used to build up the program. In this respect assembly language is the same as BASIC – a modular approach always makes programming easier.

The most complicated subroutine is f_col%, (lines 6490 to 6780). This performs the same job as the BASIC function FNC and returns the colour code of a pixel near the middle of the character location at X,Y. Most of the difficulty in writing this subroutine is due to the need to convert the text co-ordinates in X and Y to high resolution graphics co-ordinates. Notice how many instructions are needed to work out l6+32*X and 1011−32*Y as compared to BASIC!

The final program – a complete listing

Don' tworry if you haven't understood all the details of the assembly language program; it always takes a good deal more time to follow assembler than it does BASIC. If you have typed in ail the. modifications to the original Snake program you should have the following program – which will not run in the amount of memory available to it! How to make it work in so little memory is the subject of the final section of this chapter but now is the time to check that you have entered everything correctly:

```
  10 REM tadpole
  20 MODE 129
  30 AGAIN=FALSE
  40 PROCinitialise
  50 PROCtitle
  60 PROCdraw_scene
  70 REPEAT
  80 PROCone_move
  90 UNTIL NO_FOOD OR NO_LIVES
 100 PROCend_game
 110 IF AGAIN THEN GOTO 40
 120 END

1000 DEF PROCinitialise
1010 VDU 23,224,&38,&7C,&FA,&FF,&FF,&FA,&7C,&38
1020 VDU 23,225,&7E,&EF,&F7,&FB,&FB,&F7,&EF,&7E
1030 VDU 23,226,&18,&3C,&7E,&FF,&FF,&7E,&3C,&18
1040 VDU 23,227,&1C,&3E,&5F,&FF,&FF,&5F,&3F,&1C
1050 VDU 23,228,&7E,&F7,&EF,&DF,&DF,&EF,&F7,&7E
```

```
1060 VDU 23,229,&82,&54,&38,&DA,&FE,&82,&44,&82
1070 VDU 23,230,&18,&3C,&5A,&FF,&FF,&FF,&7E,&7E
1080 VDU 23,231,&7E,&FF,&E7,&D8,&BD,&FF,&FF,&7E
1090 VDU 23,232,&3C,&7E,&FF,&FF,&FF,&5A,&3C,&18
1100 VDU 23,233,&7E,&FF,&FF,&BD,&DB,&E7,&FF,&7E
1110 VDU 19,0,3,0,0,0:REM 0=YELLOW
1120 VDU 19,1,0,0,0,0:REM 1=BLACK
1130 VDU 19,2,2,0,0,0:REM 2=GREEN
1140 VDU 19,3,6,0,0,0:REM 3=CYAN
1150 IF AGAIN THEN GOTO 1260
1200 DIM X%(15),Y%(15)
1205 MAX%=15
1206 NUM%=8
1208 PROCasmb
1210 DIM T$(5)
1220 DIM T(5)
1230 FOR I=1 TO 5
1240 T(I)=0
1250 NEXT I
1260 NO_FOOD=FALSE
1270 NO_LIVES=FALSE
1280 HIT=FALSE
1290 C%=0:STOTAL%=0
1300 SC=0:SN=0:FOOD=0
1310 ENDPROC

2000 DEF PROCdraw_scene
2010 COLOUR 128+0
2020 CLS
2030 PROCdraw_snake
2040 PROCdraw_pool(18,10)
2045 COLOUR 128+3
2060 IF FOOD=0 THEN FOOD=NUM%
2070 COLOUR 1:PROCdraw_food(FOOD)
2080 COLOUR 128+0
2090 PRINT TAB(2,30);"Snake=";SN+1;" Screen=";
     SC+1;" Score=";STOTAL%+C%
2095 GCOL 0,1
2096 MOVE 0,156
2097 DRAW 1280,156
2100 TIME=0
```

```
2110 REPEAT UNTIL TIME>=RND(100)+100
2120 ENDPROC

2200 DEF PROCdraw_pool(X,Y)
2210 COLOUR 128+3
2220 PRINT TAB(X,Y);SPC(3);
2230 PRINT TAB(X-1,Y+1);SPC(5);
2240 PRINT TAB(X-2,Y+2);SPC(7);
2250 PRINT TAB(X-2,Y+3);SPC(7);
2260 PRINT TAB(X-2,Y+4);SPC(7);
2270 PRINT TAB(X-1,Y+5);SPC(5);
2280 PRINT TAB(X,Y+6);SPC(3);
2290 ENDPROC

2500 DEF PROCdraw_food(AMOUNT)
2505 IF AMOUNT=0 THEN ENDPROC
2510 FOR K=1 TO AMOUNT
2520 U%=RND(7)+15
2530 V%=RND(7)+9
2540 IF FNC(U%,V%)<>3 THEN GOTO 2520
2550 PRINT TAB(U%,V%);",";
2555 FX%?K=U%:FY%?K=V%
2560 NEXT K
2570 IF AMOUNT=NUM% THEN ENDPROC
2580 FOR K=AMOUNT+1 TO NUM%
2585 NEXT K
2590 ENDPROC

2600 DEF PROCdraw_snake
2610 COLOUR 128+0:COLOUR 1
2620 X%=2:Y%=10
2625 XH%=X%:YH%=Y%
2630 H%=224
2640 S%=225
2650 T$=CHR$(226)
2660 PRINT TAB(X%,Y%);T$;
2670 X%(1)=X%:Y%(1)=Y%
2680 FOR Q%=2 TO 10
2690 X%=X%+1
2700 PRINT TAB(X%,Y%);CHR$(S%);
2710 X%(Q%)=X%:Y%(Q%)=Y%
2720 NEXT Q%
2730 X%=X%+1
2740 PRINT TAB(X%,Y%);CHR$(H%);
```

```
2750 X%(11)=X%:Y%(11)=Y%
2760 Z%=1:Q%=11
2770 XV%=1:YV%=0
2780 ENDPROC

3000 DEF PROCone_move
3005 PROCmove_f
3010 PROCget_direction
3020 PROCupdate
3030 ENDPROC

3500 DEF PROCmove_f
3510 A%=RND(NUM%)
3520 B%=USR(move_f%)
3530 IF ?FLAG%<>0 AND FX%?A%<>&FF THEN
     PROCfrog_gone
3540 ENDPROC

3900 DEF PROCfrog_gone
3910 FX%?A%=&FF
3920 FOOD=FOOD-1
3930 IF FOOD<>0 THEN ENDPROC
3935 STOTAL%=STOTAL%+C%
3936 C%=0
3940 IF SC=4 THEN NO_FOOD=TRUE:ENDPROC
3950 SC=SC+1
3960 PROCdraw_scene
3970 ENDPROC

4000 DEF PROCget_direction
4010 IF INKEY(-58) THEN XV%=0:YV%=-1:S%=231:
     H%=230
4020 IF INKEY(-42) THEN XV%=0:YV%=1:S%=233:
     H%=232
4030 IF INKEY(-26) THEN XV%=-1:YV%=0:S%=228:
     H%=227
4040 IF INKEY(-122) THEN XV%=1:YV%=0:S%=225:
     H%=224
4045 IF ADVAL(-5)<>15 THEN GOTO 4045
4046 SOUND 0,-15,6,1
4047 SOUND 0,0,6,1
4050 ENDPROC

4500 DEF PROCupdate
4510 XH%=X%:YH%=Y%
```

```
4520 X%=X%+XV%:Y%=Y%+YV%
4530 IF X%>39 THEN X%=0
4540 IF X%<0 THEN X%=39
4550 IF Y%>26 THEN Y%=0
4560 IF Y%<0 THEN Y%=26
4570 IF FNC(X%,Y%)<>0 THEN PROChit
4580 PRINT TAB(XH%,YH%);CHR$(S%);
4590 PRINT TAB(X%,Y%);CHR$(H%);
4600 Q%=Q%+1
4610 IF Q%>MAX% THEN Q%=1
4620 X%(Q%)=X%:Y%(Q%)=Y%
4630 PRINT TAB(X%(Z%),Y%(Z%));" ";
4640 Z%=Z%+1
4650 IF Z%>MAX% THEN Z%=1
4660 PRINT TAB(X%(Z%),Y%(Z%));T$;
4670 ENDPROC

5000 DEF PROChit
5010 COL%=FNC(X%,Y%)
5020 IF COL%=3 OR COL%=1 THEN PROCtoad_self
5030 IF COL%=2 THEN PROCfrog
5040 ENDPROC

5500 DEF PROCtoad_self
5505 SOUND 1,-15,10,20:SOUND 1,0,0,10
5506 IF ADVAL(-6)<>15 THEN GOTO 5506
5510 SOUND 1,-15,10,20:SOUND 1,0,0,10
5520 SN=SN+1
5530 IF SN>4 THEN STOTAL%=STOTAL%+C%:
     NO_LIVES=TRUE:ENDPROC
5540 PROCdraw_scene
5550 ENDPROC

5600 DEF PROCfrog
5610 SOUND 1,-15,250,1
5620 C%=C%+1:FOOD=FOOD-1
5630 PRINT TAB(2,30);"Snake=";SN+1;" Screen=";
     SC+1;" Score=";STOTAL%+C%
5640 Z%=Z%-1
5650 IF Z%<1 THEN Z%=MAX%
5655 PROCrem_f
5660 IF FOOD<>0 THEN ENDPROC
5670 STOTAL%=STOTAL%+C%
5680 C%=0
5690 IF SC=4 THEN NO_FOOD=TRUE:ENDPROC
5700 SC=SC+1
```

```
5710 PROCdraw_scene
5720 ENDPROC

5800 DEF PROCrem_f
5810 K%=0
5820 REPEAT
5830 K%=K%+1
5840 UNTIL X%=FX%?K% AND Y%=FY%?K%
5850 FX%?K%=&FF
5860 ENDPROC

6000 DEF PROCasmb
6004 CODE%=&900
6005 OSWORD%=&FFF1
6010 OSWRCH%=&FFEE
6016 ATEMP%=&70
6017 FLAG%=&71
6018 PARM%=&72:COL%=PARM%+4
6019 FX%=&77:FY%=&80
6020 FOR PASS=0 TO 3 STEP 3
6030 P%=CODE%
6040 [OPT PASS
6050 .move_f% STA ATEMP%
6060 JSR  get_cords%
6070 CPX  #&FF
6080 BNE  skip1%
6090 STX  FLAG%
6100 RTS
6110 .skip1%  JSR f_col%
6120 CMP  #3
6130 BNE  skip2%
6140 LDA  #128+3
6150 JSR  s_col%
6160 .skip2% LDA COL%
6170 CMP  #1
6180 BEQ  skip3%
6190 JSR  tab%
6200 LDA  #32
6210 JSR  OSWRCH%
6220 .skip3%  JSR update%
6230 JSR  gone%
6240 BCC  skip4%
6250 STA  FLAG%
6260 RTS
6270 .skip4%  LDA #128
```

```
6280 JSR  s_col%
6290 JSR  f_col%
6300 BNE  skip3%
6310 LDA  #2
6320 JSR  s_col%
6330 JSR  tab%
6340 LDA  #229
6350 JSR  OSWRCH%
6360 LDA  #1
6370 JSR  s_col%
6380 JSR  s_cords%
6390 LDA  #0
6400 STA  FLAG%
6410 RTS
6420 \
6440 LDY  FY%,X
6450 LDA  FX%,X
6460 TAX
6470 RTS
6480 \
6500 LDA  #0
6510 STA  PARM%+1
6520 ASL  PARM%:ROL PARM%+1
6521 ASL  PARM%:ROL PARM%+1
6522 ASL  PARM%:ROL PARM%+1
6523 ASL  PARM%:ROL PARM%+1
6524 ASL  PARM%:ROL PARM%+1
6530 LDA  #16
6540 CLC
6550 ADC  PARM%
6560 STA  PARM%
6565 LDA  #0
6566 ADC  PARM%+1
6567 STA  PARM%+1
6570 STY  PARM%+2
6590 LDA  #0
6600 STA  PARM%+3
6610 ASL  PARM%+2:ROL PARM%+3
6611 ASL  PARM%+2:ROL PARM%+3
6612 ASL  PARM%+2:ROL PARM%+3
6613 ASL  PARM%+2:ROL PARM%+3
6614 ASL  PARM%+2:ROL PARM%+3
6620 LDA  #&F3
```

```
6624 SEC
6625 SBC   PARM%+2
6630 STA   PARM%+2
6631 LDA   #&3
6632 SBC   PARM%+3
6633 STA   PARM%+3
6650 TXA
6660 PHA
6670 TYA
6680 PHA
6690 LDX   #(PARM% MOD 256)
6700 LDY   #(PARM% DIV 256)
6710 LDA   #9
6720 JSR   OSWORD%
6730 PLA
6740 TAY
6750 PLA
6760 TAX
6770 LDA   PARM%+4
6780 RTS
6790 \
6801 LDA   #17
6802 JSR   OSWRCH%
6803 PLA
6804 JSR   OSWRCH%
6805 RTS
6806 \
6808 JSR   OSWRCH%
6809 TXA
6810 JSR   OSWRCH%
6811 TYA
6812 JSR   OSWRCH%
6813 RTS
6814 \
6825 BPL   u1%
6830 DEX
6835 DEX
6840 .u1% INX
6845 CPY   #13
6850 BPL   u2%
6855 DEY
6860 DEY
```

```
6865 .u2% INY
6870 RTS
6875
6882 BMI  g1%
6884 CPX  #39
6886 BPL  g1%
6888 CPY  #0
6890 BMI  g1%
6892 CPY  #26
6893 BPL  g1%
6894 CLC
6895 RTS
6896 .g1% LDA #&FF
6897 SEC
6898 RTS
6899
6901 LDX  ATEMP%
6902 STA  FX%,X
6903 TYA
6904 STA  FY%,X
6905 RTS
6910 ]
6920 NEXT PASS
6930 ENDPROC
7000 DEF PROCend_game
7010 CLS
7020 COLOUR 1
7030 IF STOTAL%<10 THEN PRINT TAB(3,10);
     "Demoted to slowworm":GOTO 7075
7040 IF STOTAL%<20 THEN PRINT TAB(3,10);
     "The frogs don't have to worry":GOTO 7075
7050 IF STOTAL%<30 THEN PRINT TAB(3,10);
     "Not bad for a grass snake":GOTO 7075
7060 IF STOTAL%<35 THEN PRINT TAB(3,10);
     "Well slithered":GOTO 7075
7070 PRINT TAB(3,10);"A venomous performance"
7074 PRINT TAB(5);"you're a SUPER SNAKE"
7075 *FX 15,0
7080 FOR I=5 TO 1 STEP -1
7090 IF STOTAL%>=T(I) THEN N=1
7100 NEXT I
7105 PRINT
7110 IF N=0 THEN PRINT TAB(10);"You scored ";
     STOTAL%:GOTO 7500
7120 PRINT TAB(10);"You are now ranked ";N
7130 PRINT TAB(10);"in the snake league"
```

```
7140 IF N=5 THEN GOTO 7200
7150 FOR I=5 TO N+1 STEP -1
7160 T(I)=T(I-1)
7170 T$(I)=T$(I-1)
7180 NEXT I
7200 T(N)=STOTAL%
7205 PRINT TAB(10);
7210 INPUT "What is your name",T$(N)
7220 CLS
7230 FOR I=1 TO 5
7240 PRINT TAB(5,I*2+5);T$(I);TAB(20);T(I)
7250 NEXT I
7500 PRINT
7510 INPUT "Another slither Y/N",A$
7520 IF LEFT$(A$,1)<>"Y" AND LEFT$(A$,1)<>"N"
     THEN GOTO 7510
7530 IF LEFT$(A$,1)="N" THEN AGAIN=FALSE ELSE
     AGAIN=TRUE
7540 ENDPROC

8000 DEF PROCtitle
8005 COLOUR 128+1:COLOUR 0
8006 CLS
8007 VDU 23,1,0;0;0;0;
8010 PROCsnake(5)
8020 PRINT TAB(2,10);
8030 PRINT "You must guide your snake using
     the"
8040 PRINT "four arrow keys and eat all the
     green"
8050 PRINT "frogs ";
8060 COLOUR 2:PRINT CHR$(229)
8070 COLOUR 0
8080 PRINT
8090 PRINT " Don't fall in the water or try to"
8120 PRINT "eat yourself or you will lose one
     of"
8130 PRINT "your five lives"
8140 PRINT
8145 PRINT
8150 PRINT "You get five screens of frogs to
     eat"
8160 PRINT "before you become a SUPER SNAKE!!"
8170 A$=INKEY$(2000)
8200 ENDPROC

8500 DEF PROCsnake(X)
8510 DATA 2,2,2,1,0,2,2,2,2,0,2,2,2,2
```

```
8520 DATA 0,7,0,7,0,2,2,2,1
8530 DATA 8,0,0,0,0,8,0,0,10,0,8,0,0,10
8540 DATA 0,8,0,8,0,8,0,0,0
8550 DATA 5,5,5,5,0,8,0,0,10,0,8,3,1,10
8560 DATA 0,8,8,0,0,8,3,2,1
8570 DATA 0,0,0,8,0,8,0,0,10,0,8,0,0,10
8580 DATA 0,8,0,8,0,8,0,0,0
8590 DATA 3,2,2,2,0,3,0,0,9,0,3,0,0,9
8595 DATA 0,3,0,3,0,5,5,5,3
8600 RESTORE
8605 PRINT TAB(X,2)
8610 FOR I=1 TO 5
8620 PRINT TAB(X);
8630 FOR J=1 TO 23
8640 READ S
8650 IF S=0 THEN PRINT " "; ELSE
     PRINT CHR$(223+S);
8660 NEXT J
8670 PRINT
8680 NEXT I
8690 ENDPROC

9000 DEF FNC(X%,Y%)
9010 X%=16+32*X%
9020 Y%=1011-32*Y%
9030 =POINT(X%,Y%)
```

Running the program

The program listed above is just too big to fit into the memory available. The principle behind making it fit into a smaller amount of space is simply to remove all the unnecessary assembly language in PROCasmb. The assembly language contained in lines 6040 to 6910 is only needed right at the start of the program when it is translated to machine code and stored in memory starting at &900. After this translation it is taking up precious memory that could otherwise be used for BASIC instructions that still have a job to do. Taking this argument one stage further it is not difficult to see that if the machine code could be loaded into memory directly, then the assembly language that produced it wouldn‹t be needed at the start of the program. This is exactly how the program is made to fit into the available memory. First, a version of the program is SAVEd which, instead of running the assembler to create the machine code, loads it into memory using *LOAD. Following this, a version of the program is produced that only calls PROCasmb and then *SAVEs the resulting machine code ready

for the first program to *LOAD.

This is a complicated procedure and so it is worth giving step-by-step instructions:

1. Save a version of the program as given in the complete listing above – this is your master copy, so keep it safe.
2. Delete lines 6020 to 6920 in PROCasmb and add the following lines:

```
6020 *LOAD MTAD
6030 move_f%=&900
6040 ENDPROC
```

   SAVE this version under the name "TADPOLE" at the start of a tape.

3. LOAD the master copy created in step 1 and make the following changes to the main program:

```
20 MODE 6
45 STOP
```

   Run the program and when it stops, type PRINT ~P%+1 and write down the result. Then type

```
*SAVE MTAD 900 x
```

   where x is the number that you wrote down. For later ease of use it is better to save the machine code version MTAD on the same tape that you used in step 2, following TADPOLE as closely as possible.

4. Now you are ready to run the final version of Tadpole. Rewind the tape generated in step 3; it should have both TADPOLE and MTAD on it. Then type CHAIN "TADPOLE" and the BASIC part of the program will load and begin running. As one of its first tasks this will load the machine code in MTAD and this is the reason that it is convenient to have the file MTAD following TADPOLE.

This procedure may sound a little complicated but once the tape has been made up. the program can be loaded and run as easily as any mber. The finishing touch would be to suppress the tape loading message (by using OPT) so the user didn't even know that the second file was being loaded!

Conclusion

This chapter has illustrated some of the problems inherent in trying to produce large programs that use assembler when memory is limited. If you feel like trying to extend the program any further then your biggest headache will most certainly be how to fit any more into the space left over. Perhaps at this stage you might consider converting the entire program into assembler. If you prefer a little less work, an alternative solution is to abandon Mode 1 for the two-colour Mode 4; not such a pretty display but a much easier program.

Chapter Seven
Snakes and Ladders

In this chapter a traditional and well known game, Snakes and Ladders, is given new life in a computer implementation. Traditional games are an obvious source for anyone looking for new ideas for computer games. Not only can they be implemented according to their original rules, they often provide the starting point for games that could only be played with the aid of a computer. As an example of this idea, at the end of this chapter Snakes and Ladders is modified into a fast action computer game that could never have been played using a board and dice!

Snakes and Ladders is different from all the other games in this book in that it apparently doesn't involve animation. It is true that the main difficulty in implementing Snakes and Ladders lies in the construction of the graphics and the board, however many of the techniques of animation introduced in earlier chapters prove useful.

The game design

The form and rules of Snakes and Ladders are known to nearly everyone and the only real issues of game design concern how it should be transferred to computer. The traditional version of the game consists of a board divided into squares with snakes and ladders connecting squares in different rows. The game is usually played by more than one person at a time and the object is to be first to move a counter from the first square at the bottom left of the board to the final square at the top left. The players move alternately and the number of squares moved is governed by the fall of a dice – if a move lands a player on a snake's head then the player moves backwards to the position occupied by the snake's tail. Similarly if a move ends at the foot of a ladder then the player moves forward to the square containing the top of the ladder. In other words, if you land on the head of a snake you slide down it but if you land at the foot of a ladder you climb up it.

In the computer version of the game it would add to the interest to have the board created anew with a random layout of snakes and ladders each time the game is played. Also, the current position of each player could be marked using a man-shaped graphics character and this suggests various possibilities for animating the character sliding down snakes and climbing up ladders! Other obvious tasks for the program are keeping track of whose turn it is and even generating random numbers to eliminate the need for a dice!

As already mentioned, the main challenge in implementing Snakes and Ladders is in designing and handling the graphics that make up the individual snakes and ladders. An example of a Snakes and Ladders board generated by the program can be seen in Fig. 7.1. The snakes and the ladders are produced using a number of different graphics characters printed in combination. Not only are the starting positions of the Snakes and Ladders random but so are their lengths.

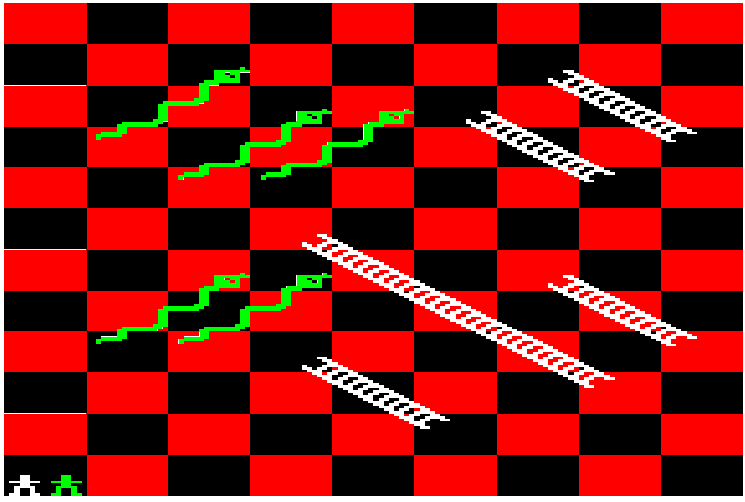


Fig. 7.1.

The only real problem in constructing the board is making sure that snakes and ladders do not ‘clash’ that is, cross over each other. There are two main ways of achieving this separation. Firstly, the position of the start and end of every snake and every ladder could be recorded in an array and every time something was about to be added to the board the array could be examined to see if it overlapped with anything already on the board. The second method is simply to examine every

chacter position on the screen where a new object is going to be printed to make sure that they are all blanks. In practice, unless speed is of extreme importance it is always simpler to examine what is already on screen. However, although this method is simple in theory there are a number of practical difficulties to be overcome and these are described in later sections. The same method of examining which character is already printed on the screen can be used to discover if a player has landed on snake's head of the foot of a ladder. In fact you could say that the whole basis of the Snakes and Ladders program is finding out what character is already printed on the screen!

The main program

Although Snakes and Ladders isn't a dynamic graphics game its main program still has the (by now) familiar form:

```
 10 REM Snakes and Ladders
 20 MODE 4
 30 PROCinit
 40 PROCtitle
 50 MODE 5:PROCcolour
 60 PROCp_sandl
 70 PROCstart
 80 REPEAT
 90 I%=0:PROCthrow(I%)
100 IF NOT HOME THEN I%=1:PROCthrow(I%)
110 UNTIL HOME
120 PROCendgame
130 IF AGAIN=TRUE THEN RUN
140 END
```

Mode 5 is used (line 50) for Snakes and Ladders because it certainly needs four colours and the board can be made to fit into the 20 by 32 screen. Indeed in this case the large size of the characters in mode 5 is a positive advantage in producing a bold Snakes and Ladders board. However, mode 4 (line 20) is used to print the title frame because it is difficult to produce good looking text using only 20 columns. The use of two different modes in the program means that it is necessary to set the logical to physical colour in a procedure that is separate from PROCinit. After mode 5 is selected line 60 calls PROCp_sandl which prints the Snakes and Ladders board on the screen. Following this,

PROCstart is called to initialise some of the variables for running the game. Each time through the loop both players make one move. Line 90 calls PROCthrow to allow player zero a chance to move and line 100 calls PROCthrow to allow player one a chance to move. The variable HOME is used to indicate that one of the players has reached the top left hand corner of the board. Finally, when the game is over line 120 calls PROCendgame and line 130 either restarts the entire game or brings it to an end depending on the state of AGAIN.

PROCinit, PROCtitle and PROCcolour

PROCinit simply sets up the graphics characters and arrays used in Snakes and Ladders:

```
1000 DEF PROCinit
1010 VDU 23,224,&00,&02,&7F,&6C,&74,&7C,&FC,&C0
1020 VDU 23,225,&03,&03,&03,&03,&03,&07,&FE,&FC
1030 VDU 23,226,&00,&00,&00,&00,&00,&00,&01,&03
1040 VDU 23,227,&03,&03,&1F,&3E,&20,&00,&00,&00
1050 VDU 23,228,&00,&00,&06,&03,&01,&23,&36,&1C
1060 VDU 23,229,&98,&3C,&66,&CF,&D9,&73,&36,&1C
1070 VDU 23,230,&00,&00,&00,&00,&80,&C0,&60,&F0
1080 VDU 23,231,&0D,&07,&03,&01,&00,&00,&00,&00
1090 VDU 23,232,&98,&3C,&66,&C0,&C0,&60,&00,&00
1100 VDU 23,233,&18,&18,&7E,&18,&3C,&24,&24,&66
1150 DIM VX%(1),N$(1)
1160 DIM X%(1),Y%(1),Q%(1)
1170 ENDPROC
```

The way the graphics characters go together to make a snake or a ladder is quite complicated and can be best understood from Fig. 7.2. The arrays X% and Y% are used to hold the co-ordinate of the man shape CHR$(233) for each player. What the arrays VX% and Q% are used for is easier to describe later on along with the method of making each move. The string array N$ is used to hold the names of each of the players: N$(0) holds the name of player zero and N$(1) holds the name of player one.

PROCtitle prints instructions about how to play the game and also asks the players for their names and whether they want to use a real dice or not.

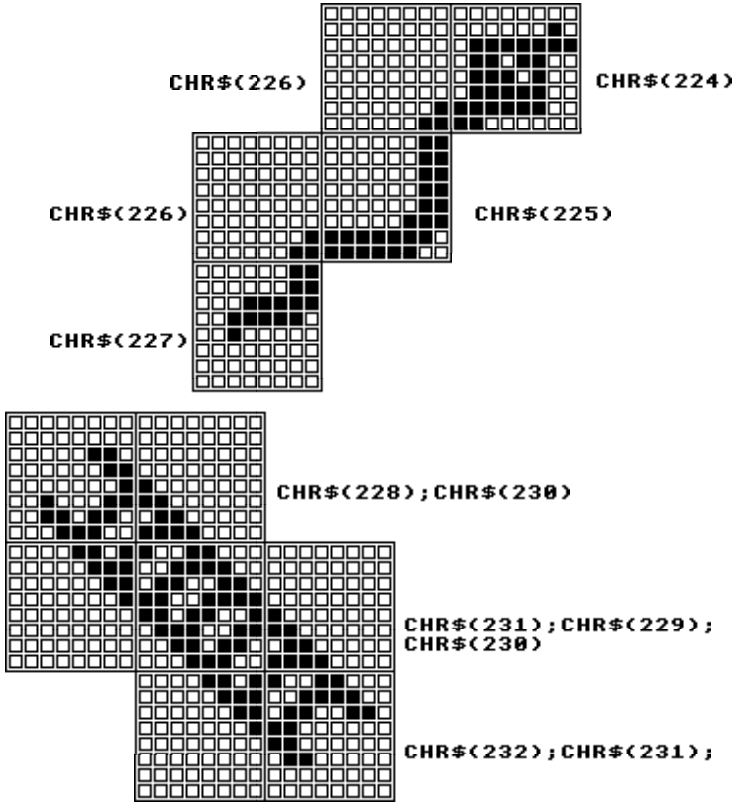


Fig. 7.2. Graphics characters for (a) snake and (b) ladder

```
8000 DEF PROCtitle
8010 COLOUR 128+0
8020 COLOUR 1
8025 CLS
8030 PRINT TAB(3,3)"S N A K E S  A N D
     L A D D E R S"
8040 PRINT TAB(0,10);
8050 PRINT TAB(3);"This is a game for two
     players"
8060 PRINT TAB(2);"The first player to reach
     the top"
8070 PRINT TAB(2);"left of the board wins!!"
8080 PRINT TAB(0,20);
```

```
8090 INPUT "Do you want to use a real dice",A$
8100 A$=LEFT$(A$,1)
8110 IF A$<>"Y" AND A$<>"N" THEN GOTO 8010
8120 IF A$="Y" THEN DICE=TRUE ELSE DICE=FALSE
8140 PRINT TAB(0,25);
8150 INPUT "What is the first players
     name",N$(0)

8155 PRINT TAB(0,27)
8160 INPUT "What is the second players
     name",N$(1)
8170 ENDPROC
```

Lines 8080 to 8120 set the variable DICE according to whether the players want to use a real dice (DICE=TRUE) or have the computer generate random numbers (DICE=FALSE). Lines 8150 to 8160 get the players' names and store them into N$(0) and N$(1).

In the earlier games, PROCcolour has been part of PROCinit because there was no reason to set the logical to physical colours apart from at the very start of the game.

```
1500 DEF PROCcolour
1510 VDU 19,0,0,0,0,0:REM 0=BLACK
1520 VDU 19,1,1,0,0,0:REM 1=RED
1530 VDU 19,2,2,0,0,0:REM 2=GREEN
1540 VDU 19,3,7,0,0,0:REM 3=WHITE
1550 ENDPROC
```

Black and red are used for the board, green is used for the snakes and one of the men, and white is used for the ladders and the other man.

Printing the board – PROCp_sandl and its associated procedures

The purpose of PROCp_sandl is to print the board complete with snakes and ladders. It does this mainly by calls to other procedures:

```
2500 DEF PROCp_sandl
2510 PROCp_squares
2520 FOR S%=1 TO 3+RND(2)
2530 COLOUR 2:PROCmake_snake
2540 COLOUR 3:PROCmake_ladder
2545 NEXT S%
2550 ENDPROC
```

PROCp_squares prints the pattern of black and red squares that makes up the board. PROCmake_snake and PROCmake_ladder each print a single snake and a single ladder respectively. The FOR loop, lines 2520 to 2545, calls PROCmake_snake and PROCmake_ladder a random number of times between 3 and 5.

PROCp_squares is fairly straightforward:

```
2000 DEF PROCp_squares
2010 COLOUR 128
2020 CLS
2024 VDU 23,1,0;0;0;0;
2025 C%=0
2030 FOR Y%=1 TO 23 STEP 2
2040 FOR X%=1 TO 18 STEP 2
2045 COLOUR 129+C%
2046 C%=NOT C%
2050 PRINT TAB(X%,Y%);SPC(2);
2051 PRINT TAB(X%,Y%+1);SPC(2);
2060 NEXT X%
2070 NEXT Y%
2080 ENDPROC
```

Each of the squares that make up the board is composed of four blanks. These are alternately printed in black and red by using the variable C% to set the colour (in line 2045).

PROCmake_snake looks a little complicated but this is only because of the large number of PRINT statements necessary to print a snake:

```
3000 DEF PROCmake_snake
3005 TRY=0
3010 X%=RND(6)+2:Y%=RND(9)
3015 IF X%-1<(10-Y%) THEN L%=RND(X%-2)+1
     ELSE L%=RND(9-Y%)+1
3016 IF TRY>20 THEN ENDPROC ELSE TRY=TRY+1
3020 IF FNclash(X%,Y%,L%,-1) THEN GOTO 3010
3040 PROCprint(X%*2,Y%*2,224)
3045 PROCprint(X%*2-1,Y%*2,226)
3050 FOR Z%=X%-1 TO X%-L%+2 STEP -1
3055 Y%=Y%+1
3060 PROCprint(Z%*2+1,Y%*2-1,225)
3065 PROCprint(Z%*2,Y%*2-1,226)
3070 PROCprint(Z%*2,Y%*2,225)
3075 PROCprint(Z%*2-1,Y%*2,226)
```

```
3080 NEXT Z%
3085 Y%=Y%+1
3086 PROCprint(Z%*2+1,Y%*2-1,227)
3090 ENDPROC
```

Line 3010 generates two random numbers that specify the starting position of a snake. Line 30 15 then generates another random number in L% that determines the length of the snake. The IF statement makes sure that L% is not so large that the snake goes off the edge of the board. Line 3020 uses the function FN clash to check whether or not the snake specified by X%, Y% and L% crosses any other snake or a ladder. If it does, then control is passed back to line 3010 and another set of random values of X%, Y% and L% are tried. In this way various starting positions and lengths of snake are tried until one is found that fits. that is, one that can be printed without overlaying any snakes or ladders that are already on the board. As there is no guarantee that a snake that fits will be found in a reasonable amount of time, the variable TRY is used to count the number of attempts. Line 3016 makes PROCmake_snake give up after 20 tries. Once a snake that fits has been found, lines 3040 to 3086 print the combination of characters to make it appear on the board. PROCprint is used rather than a simple print statement because of the need to select the correct background colour to match the colour already present in the character location. That is, where the snake is positioned in a red square, PROCprint will use a red background and where it is positioned in a black square PROCprint will use a black background.

PROCmake ladder works in roughly the same way as PROC make_snake:

```
3500 DEF PROCmake_ladder
3505 TRY=0
3510 X%=RND(7):Y%=RND(9)
3515 IF TRY>20 THEN ENDPROC ELSE TRY=TRY+1
3520 IF 9-X%<11-Y% THEN L%=(9-X%) ELSE
     L%=(11-Y%)
3530 IF FNclash(X%,Y%,L%,1) THEN GOTO 3510
3540 PROCprint(X%*2,Y%*2,228)
3541 PROCprint(X%*2+1,Y%*2,230)
3542 PROCprint(X%*2,Y%*2+1,231)
3546 Y%=Y%+1
3550 FOR Z%=X%+1 TO X%+L%-2
3555 PROCprint(Z%*2-1,Y%*2-1,229)
3560 PROCprint(Z%*2,Y%*2,229)
```

```
3565 PROCprint(Z%*2+1,Y%*2,230)
3566 PROCprint(Z%*2,Y%*2+1,231)
3567 PROCprint(Z%*2,Y%*2-1,230)
3568 PROCprint(Z%*2-1,Y%*2,231)
3580 Y%=Y%+1
3590 NEXT Z%
3610 PROCprint(Z%*2-1,Y%*2-1,232)
3630 ENDPROC
```

Lines 3505 to 3530 find values of X%, Y%, and L% that produce a ladder that doesn't interfere with any ladders or snakes already printed on the screen. Lines 3540 to 3610 print the combination of characters that produces a ladder.

FNclash works by examining what is already printed on the screen in every character location that would be affected by printing a ladder or a snake.

```
9000 DEF FNclash(X%,Y%,L%,D%)
9005 LOCAL T%
9006 T%=FALSE
9007 E%=2*(X%+D%*L%)+D%:X%=X%*2-D%:Y%=Y%*2-1
9010 REPEAT
9020 IF FNchar(X%,Y%)<>32 THEN T%=TRUE
9025 IF FNchar(X%+1,Y%)<>32 THEN T%=TRUE
9026 IF FNchar(X%-1,Y%)<>32 THEN T%=TRUE
9030 X%=X%+D%:Y%=Y%+1
9040 UNTIL X%=E% OR T%=TRUE
9050 =T%
```

Line 9007 works out which character locations have to be examined. Lines 9010 to 9040 then proceed to examine each location in turn until either the last location has been examined or until a non-blank character is found. Lines 9020 to 9026 do the actual examination of what is already printed at each character location using another function, FNchar. FNchar(X%,Y%) returns the ASCII code of the character printed at X%, Y%. However, for the user-defined graphics characters corresponding to ASCII codes 224 to 233 you will discover that FNchar returns codes in the range 128 to 137. The reason for this is that, unless memory is set aside for more than the standard set of user defined characters. ASCII codes 128, 160. 192 and 224 all produce the same character. Similarly. ASCII codes 129, 161, 193 and 225 all produce the same character and so on. The function FNchar simply returns the lowest ASCII value that will produce the churacter on the screen at

X%,Y% and this has to be allowed for when testing for the presence of user defined characters (see later).

FNchar is based on the function given in the User Guide in connection with OSBYTE 135.

```
9700 DEF FNchar(X%,Y%)
9710 LOCAL A%
9715 COLOUR 128+FNC(X%,Y%)
9720 VDU 31,X%,Y%
9730 A%=135
9740 =(USR(&FFF4) AND &FF00) DIV &100
```

Line 9720 moves the text cursor to X%, Y% and then lines 9730 and 9740 simply make a call to OSBYTE with A set to 135. OSBYTE 135 will return the ASCII code of the character printed at X%,Y%, or 0 if the character is not recognisable. Line 9715 uses the, by now, familiar function FNC to set the background colour to whatever colour is already on the screen at X%,Y%

The reason why the background colour has to be set in this way is u consequence of the way OSBYTE 135 works. OSBYTE 135 reads the colours of each of the pixels in the eight-by-eight character location at X%,Y% and classifies them into foreground and background pixels, All the pixels on the screen that are the same colour as the background colour, as set by the last COLOUR statement, are classified as background points and the rest are classified as foreground points. This means that if the background colour has changed since the character was printed, OSBYTE 135 may not be able to recognise it. For example, if a space character, ASCII code 32, is printed using background colour 129 and OSBYTE 135 is immediately called to identify it, then all of the pixels in the eight-by-eight character location will be classified as background pixels and OSBYTE 135 will match it with the space character in other words it will return 32. However, if the background colour is changed to 130 after the space is printed and then OSBYTE 135 is called, all of the pixels will be classified as foreground pixels and, unless there is a ‘solid block’ character defined, OSBYTE 135 will fail to recognise it and return 0 as the result. FNchar avoids this problem by always setting the background colour to the colour of one of the pixels within the character location. This trick only works if the pixel examined is always a background pixel and this is where FNC comes in:

```
9500 DEF FNC(X%,Y%)
9510 X%=8+64*X%
```

```
9520 Y%=1023-32*Y%
9530 =POINT(X%,Y%)
```

This version of FNC the pixel that is examined is the second pixel in the top row of the character location at X%,Y% and if you examine all the user-defined character definitions you will find that this pixel is a background pixel in all of them!

PROCprint also uses FNC to set the background colour to the colour that is already on the screen:

```
9600 DEF PROCprint(X%,Y%,C%)
9610 COLOUR 128+FNC(X%,Y%)
9620 PRINT TAB(X%,Y%);CHR$(C%);
9630 ENDPROC
```

This procedure will print CHR$(C%) at character location X%,Y% using the background colour that is already present.

There is another and simpler way of achieving the same result, by using VDU 5. This VDU code causes the text screen and the high resolution graphics to behave in the same way. For example, following VDU 5 the position at which text appears on the screen is controlled by the graphics cursor rather than the text cursor. A side effect of VDU 5 is that text can be printed in such a way that it 'overlays' rather than replaces what is already on the screen. In other words, you can automatically preserve the background colour following VDU 5. However, the difficulties involved in working with high resolution graphics co-ordinates when printing text characters is something worth avoiding if at all possible! There is also the disadvantage that printing takes longer after a VDU 5 code. All in all, PROCprint seems like a good way of preserving the background colour without too much difficulty.

PROCstart

PROCstart initialises the arrays used later in the program and prints the two man shapes at their starting positions:

```
3700 DEF PROCstart
3710 FOR I%=0 TO 1
3720 X%(I%)=1:Y%(I%)=12
3730 Q%(I%)=32:VX%(I%)=+1
```

```
3735 COLOUR 2+I%
3736 PROCprint(X%(I%)*2-I%,Y%(I%)*2,233)
3740 NEXT I%
3750 HOME=FALSE
3770 ENDPROC
```

As already mentioned, X% and Y% are used to hold the positions of the two players' man characters. Although each square of the board is composed of four character locations the positions are recorded so that X%(I%)*2,Y%(I%)*2 is the bottom right hand character of the square in the X%(I%)th column and Y%(I%)th row of the board. The array VX% is used to store the direction of motion of each man character. At each update the position of the I% player's man is changed to:

```
X%(I%)=X%(I%)+VX%(I%)
Y%(I%)=Y%(I%)-1
```

The array Q% is used to store the ASCII code of the blanking character to be used at the next move of the I%th man character. In all the earlier games, characters that moved did so against a clear background and so each time through the animation loop it could be blanked out using n space character of the appropriate colour. However, the man characters in Snakes and Ladders move on a background that consist of a wide range of different characters. In this case it is necessary to save the character that is present in the character location before one of the man characters is printed, and restore it when the man character moves to another location. In other words, the character used to blank a man character at its old position depends on what was present at the location before the man was printed. As the initial position of the man characters is a blank square, Q%(I%) is initialised to 32, the ASCII code for a space, by line 3730.

Moving men – PROCthrow and its associated procedures

Once the board is printed and the man characters set up at their starting positions, PROCthrow is called repeatedly to move them around the board:

```
5000 DEF PROCthrow(I%)
5010 IF DICE THEN PROCdice1 ELSE PROCdice2
5020 REPEAT
5030 PROCone_move(I%)
```

```
5035 SOUND 1,-15,80+I%*8,2
5040 M%=M%-1
5050 HOME%=(X%(I%)=1 AND Y%(I%)=1)
5060 UNTIL M%=0 OR HOME
5065 IF SNAKE AND LADDER THEN PROCchoose
5070 IF SNAKE THEN PROCupdown(I%,+1,131)
5080 IF LADDER THEN PROCupdown(I%,-1,132)
5090 HOME=(X%(I%)=1 AND Y%(I%)=1)
5100 ENDPROC
```

Depending on the value of DICE, line 3010 calls one of the two dice procedures either to ask player 195 the result of throwing a real dice PROCdice1) or to generate a random number to simulate the throw of a dice (PROCdice2). No matter which dice procedure is called, the result is returned in M% and lines 5030 to 5060 then move man 195 by M% squares. A move of a single square is produced by calling PROCone_move (line 3030) and line 5050 checks to see if the move has resulted in the man reaching the top right hand square. After each call ROCone_move sets SNAKE to TRUE if the man has landed on a snake's head and LADDER to TRUE if the man has landed at the foot of a ladder. Lines 5065 to 5080 check for the man landing on a snake or a ladder as a result of the last move. Line 5065 resolves the problem caused by landing on a square that contains both a snake's head and the foot of a ladder by calling PROCchoose, which chooses either the ladder or the snake at random. PROCupdown deals with making the man either slide down a snake or climb up a ladder (lines 5070 and 5080).

PROCone move has three different tasks to carry out. It has to look after the blanking of the man character at its old position, update the co-ordinates taking account of what happens when the man reaches the edge of the board and test for the presence of a snake's head of the start of a ladder within the new square.

```
4000 DEF PROCone_move(I%)
4010 PROCblank(I%,X%(I%)*2-I%,Y%(I%)*2)
4020 X%(I%)=X%(I%)+VX%(I%)
4030 IF X%(I%)>9 THEN X%(I%)=9:
     Y%(I%)=Y%(I%)-1:VX%(I%)=-VX%(I%)
4040 IF X%(I%)<1 THEN X%(I%)=1:
     Y%(I%)=Y%(I%)-1:VX%(I%)=-VX%(I%)
4050 Q%(I%)=FNchar(X%(I%)*2-I%,Y%(I%)*2)
4055 IF FNchar(X%(I%)*2,Y%(I%)*2)=128 THEN
     SNAKE=TRUE ELSE SNAKE=FALSE
4056 IF FNchar(X%(I%)*2-1,Y%(I%)*2-1)=136
     THEN LADDER=TRUE ELSE LADDER=FALSE
4060 COLOUR 2+I%
```

```
4070 PROCprint(X%(I%)*2-I%,Y%(I%)*2,233)
4080 ENDPROC
```

The blanking of the man character at its old position is taken care of by line 4010 which calls PROCblank antd line 4050 which saves the ASCII code of the character currently on the screen in Q%(I%) ready to be used the next time PROCone_move is called. The update of the co-ordinates of the man character is implemented by lines 4020 to 4040. You should be able to recognise lines 4030 and 4040 as a sort of 'bounce' off the edge of the board. That is, on reaching the edge of the board the man moves up by one row (i.e. Y%(I%)=Y%(I%)−1) and its direction of motion is reversed (i.e. VX%(I%)=−VX%(I%)). Finally, lines 4053 and 4056 check for a snake's head and the foot of a ladder respectively and then line 4070 prints the man at its new position.

This leaves a number of small procedures and one large procedure, PROCupdown, to describe. Before moving on to PROCupdown it is better to deal with the smaller procedures. PROCdice1 asks for the result of throwing a real dice:

```
5500 DEF PROCdice1
5501 COLOUR 128:COLOUR 2+I%
5505 PRINT TAB(0,27);SPC(60);
5510 PRINT TAB(0,27);"It's your throw"
5520 PRINT " -";N$(I%)
5530 INPUT "What did you get ",M%
5540 IF M%<1 OR M%>6 THEN GOTO 5510
5550 ENDPROC
```

PROCdice2 uses the RND function to supply a random number instead of throwing a real dice:

```
5600 DEF PROCdice2
5601 COLOUR 128:COLOUR 2+I%
5610 PRINT TAB(0,28);SPC(60);
5620 PRINT TAB(0,28);"It's your throw"
5630 PRINT " -";N$(I%)
5640 PRINT "PRESS ANY KEY"
5645 *FX 15,0
5650 REPEAT
5660 M%=RND(6)
5670 PRINT TAB(0,31);M%;
5680 UNTIL INKEY(0)<>-1
5690 ENDPROC
```

Lines 5650 to 5680 print random numbers in the range 1 to 6 until the player presses a key.

PROCchoose sets one of SNAKE or LADDER to FALSE at random so choosing one of the two possibilities:

```
5200 DEF PROCchoose
5210 IF RND(1)>.5 THEN LADDER=FALSE ELSE
     SNAKE=FALSE
5220 ENDPROC
```

Finally PROCblank(I%,X%,Y%) will print the original character at X%,Y% to blank out player I%'s man character:

```
4500 DEF PROCblank(I%,X%,Y%)
4510 IF Q%(I%)>127 AND Q%(I%)<132 THEN COLOUR 2
4520 IF Q%(I%)>131 AND Q%(I%)<137 THEN COLOUR 3
4525 IF Q%(I%)=137 THEN COLOUR 3-I%
4530 PROCprint(X%,Y%,Q%(I%))
4540 ENDPROC
```

Lines 4510 to 4525 select the correct foreground colour for the character, green for part of a snake, white for part of a ladder and the appropriate colour for either of the man characters. Then line 4530 uses PROCprint to restore the character using the correct background colour.

The final procedure concerned with moving the man characters is PROCupdown. This makes the man either slide down a snake or climb up a ladder:

```
6000 DEF PROCupdown(I%,D%,C%)
6010 LOCAL X%,Y%
6020 PROCblank(I%,X%(I%)*2-I%,Y%(I%)*2)
6030 X%=X%(I%)*2:Y%=Y%(I%)*2
6040 REPEAT
6050 Q%(I%)=FNchar(X%,Y%)
6055 COLOUR 2+I%
6060 PROCprint(X%,Y%,233)
6065 SOUND 1,-15,50-2*Y%,2
6070 PROCblank(I%,X%,Y%)
6080 X%=X%-1:Y%=Y%+D%
6090 UNTIL Q%(I%)=C% OR FNon_man
6095 X%=X%+1:Y%=Y%-D%
6100 PROCblank(I%,X%,Y%)
6110 X%(I%)=X% DIV 2
6120 Y%(I%)=Y% DIV 2
6125 IF D%=1 THEN Y%(I%)=Y%(I%)+1:
     X%(I%)=X%(I%)+1
```

```
6130 Q%(I%)=FNchar(X%(I%)*2-I%,Y%(I%)*2)
6135 COLOUR 2+I%
6140 PROCprint(X%(I%)*2-I%,Y%(I%)*2,233)
6150 IF (Y%(I%) DIV 2)*2=Y%(I%) THEN
     VX%(I%)=1 ELSE VX%(I%)=-1
6160 ENDPROC
```

The way PROCupdown works is simply by printing the man character at each position on the snake or the ladder using PROCblank to blank out the old version of the man (lines 6020, 6070 and 6100), making appropriate sound effects on the way (line 6065), until the character that marks the end of the snake or ladder is found (line 6090). The value Of D9% controls whether the man is moved up a ladder (D%=1) or down a snake (D%=–1) and C% is the ASCII code of the final character of a ladder or of a snake. The only complication is that there may not be a final character on the ladder that the man is climbing up. The reason for this is that the other man character may be positioned anywhere along the ladder just because the ladder happens to pass through the square that the man occupies. To deal with this, FNon_man is called to examine the character that originally occupied the location:

```
6500 DEF FNon_man
6510 LOCAL J%
6515 IF Q%(I%)<>137 THEN =FALSE
6520 IF I%=1 THEN J%=0 ELSE J%=1
6530 IF Q%(J%)=132 THEN =TRUE ELSE=FALSE
```

This concludes the description of the move logic part of Snakes and Ladders and it has to be admitted that much of it was developed as a result of problems that arose during testing. For example, the situation Where one man is climbing a ladder that the other man is already at the top of was not foreseen in the early planning of the program and FNon_man was added as an afterthought. This is in fact very much the way that the development of a program in which many different things can happen at random often proceeds. It is almost impossible to be aware of every rare event that might happen before a program is written but as long as you have used a modular structure then it should be easy to add new procedures to cope with situations as they arise.

PROCendgame

PROCendgame simply congratulates the player who is first home and asks if another game is required:

```
8500 DEF PROCendgame
8505 *FX 15,0
8510 COLOUR 128:COLOUR 2+I%
8520 PRINT TAB(0,27);SPC(99);
8525 PRINT TAB(0,27);
8530 PRINT "You win ";N$(I%)
8535 PRINT TAB(0,28);
8540 INPUT "Another game ",A$
8550 A$=LEFT$(A$,1)
8560 IF A$<>"Y" AND A$<>"N" THEN GOTO 8535
8570 IF A$="Y" THEN AGAIN=TRUE ELSE AGAIN=FALSE
8580 ENDPROC
```

Conclusion

The final program works very quickly and for a mode 5 program has good and effective graphics. There is still plenty of scope for improvement and plenty of memory left to do it in! For example, the game seems to encourage more personal involvement when a real dice is used in place of the random number generator. This suggests that the addition of a simulated dice on the screen might improve the game. The biggest problem with the game is that the outcome of the game is entirely a matter of chance and while this is fun, and even interesting for a while, it doesn't really present any challenge or require any skill to play. This problem can only be dealt with by changing the nature of the game quite a lot. The final part of this chapter gives the modifications necessary for one such new game. Meanwhile, however, the computer version of the traditional Snakes and Ladders game is ready to be played.

The complete listing

```
10 REM Snakes and Ladders
20 MODE 4
30 PROCinit
40 PROCtitle
50 MODE 5:PROCcolour
```

```
  60 PROCp_sandl
  70 PROCstart
  80 REPEAT
  90 I%=0:PROCthrow(I%)
 100 IF NOT HOME THEN I%=1:PROCthrow(I%)
 110 UNTIL HOME
 120 PROCendgame
 130 IF AGAIN=TRUE THEN RUN
 140 END

1000 DEF PROCinit
1010 VDU 23,224,&00,&02,&7F,&6C,&74,&7C,&FC,&C0
1020 VDU 23,225,&03,&03,&03,&03,&03,&07,&FE,&FC
1030 VDU 23,226,&00,&00,&00,&00,&00,&00,&01,&03
1040 VDU 23,227,&03,&03,&1F,&3E,&20,&00,&00,&00
1050 VDU 23,228,&00,&00,&06,&03,&01,&23,&36,&1C
1060 VDU 23,229,&98,&3C,&66,&CF,&D9,&73,&36,&1C
1070 VDU 23,230,&00,&00,&00,&00,&80,&C0,&60,&F0
1080 VDU 23,231,&0D,&07,&03,&01,&00,&00,&00,&00
1090 VDU 23,232,&98,&3C,&66,&C0,&C0,&60,&00,&00
1100 VDU 23,233,&18,&18,&7E,&18,&3C,&24,&24,&66
1150 DIM VX%(1),N$(1)
1160 DIM X%(1),Y%(1),Q%(1)
1170 ENDPROC

1500 DEF PROCcolour
1510 VDU 19,0,0,0,0,0:REM 0=BLACK
1520 VDU 19,1,1,0,0,0:REM 1=RED
1530 VDU 19,2,2,0,0,0:REM 2=GREEN
1540 VDU 19,3,7,0,0,0:REM 3=WHITE
1550 ENDPROC

2000 DEF PROCp_squares
2010 COLOUR 128
2020 CLS
2024 VDU 23,1,0;0;0;0;
2025 C%=0
2030 FOR Y%=1 TO 23 STEP 2
2040 FOR X%=1 TO 18 STEP 2
2045 COLOUR 129+C%
2046 C%=NOT C%
2050 PRINT TAB(X%,Y%);SPC(2);
2051 PRINT TAB(X%,Y%+1);SPC(2);
2060 NEXT X%
```

```
2070 NEXT Y%
2080 ENDPROC

2500 DEF PROCp_sandl
2510 PROCp_squares
2520 FOR S%=1 TO 3+RND(2)
2530 COLOUR 2:PROCmake_snake
2540 COLOUR 3:PROCmake_ladder
2545 NEXT S%
2550 ENDPROC

3000 DEF PROCmake_snake
3005 TRY=0
3010 X%=RND(6)+2:Y%=RND(9)
3015 IF X%-1<(10-Y%) THEN L%=RND(X%-2)+1
     ELSE L%=RND(9-Y%)+1
3016 IF TRY>20 THEN ENDPROC ELSE TRY=TRY+1
3020 IF FNclash(X%,Y%,L%,-1) THEN GOTO 3010
3040 PROCprint(X%*2,Y%*2,224)
3045 PROCprint(X%*2-1,Y%*2,226)
3050 FOR Z%=X%-1 TO X%-L%+2 STEP -1
3055 Y%=Y%+1
3060 PROCprint(Z%*2+1,Y%*2-1,225)
3065 PROCprint(Z%*2,Y%*2-1,226)
3070 PROCprint(Z%*2,Y%*2,225)
3075 PROCprint(Z%*2-1,Y%*2,226)
3080 NEXT Z%
3085 Y%=Y%+1
3086 PROCprint(Z%*2+1,Y%*2-1,227)
3090 ENDPROC

3500 DEF PROCmake_ladder
3505 TRY=0
3510 X%=RND(7):Y%=RND(9)
3515 IF TRY>20 THEN ENDPROC ELSE TRY=TRY+1
3520 IF 9-X%<11-Y% THEN L%=(9-X%) ELSE
     L%=(11-Y%)
3530 IF FNclash(X%,Y%,L%,1) THEN GOTO 3510
3540 PROCprint(X%*2,Y%*2,228)
3541 PROCprint(X%*2+1,Y%*2,230)
3542 PROCprint(X%*2,Y%*2+1,231)
3546 Y%=Y%+1
3550 FOR Z%=X%+1 TO X%+L%-2
3555 PROCprint(Z%*2-1,Y%*2-1,229)
```

```
3560 PROCprint(Z%*2,Y%*2,229)
3565 PROCprint(Z%*2+1,Y%*2,230)
3566 PROCprint(Z%*2,Y%*2+1,231)
3567 PROCprint(Z%*2,Y%*2-1,230)
3568 PROCprint(Z%*2-1,Y%*2,231)
3580 Y%=Y%+1
3590 NEXT Z%
3610 PROCprint(Z%*2-1,Y%*2-1,232)
3630 ENDPROC

3700 DEF PROCstart
3710 FOR I%=0 TO 1
3720 X%(I%)=1:Y%(I%)=12
3730 Q%(I%)=32:VX%(I%)=+1
3735 COLOUR 2+I%
3736 PROCprint(X%(I%)*2-I%,Y%(I%)*2,233)
3740 NEXT I%
3750 HOME=FALSE
3770 ENDPROC

4000 DEF PROCone_move(I%)
4010 PROCblank(I%,X%(I%)*2-I%,Y%(I%)*2)
4020 X%(I%)=X%(I%)+VX%(I%)
4030 IF X%(I%)>9 THEN X%(I%)=9:
     Y%(I%)=Y%(I%)-1:VX%(I%)=-VX%(I%)
4040 IF X%(I%)<1 THEN X%(I%)=1:
     Y%(I%)=Y%(I%)-1:VX%(I%)=-VX%(I%)
4050 Q%(I%)=FNchar(X%(I%)*2-I%,Y%(I%)*2)
4055 IF FNchar(X%(I%)*2,Y%(I%)*2)=128 THEN
     SNAKE=TRUE ELSE SNAKE=FALSE
4056 IF FNchar(X%(I%)*2-1,Y%(I%)*2-1)=136
     THEN LADDER=TRUE ELSE LADDER=FALSE
4060 COLOUR 2+I%
4070 PROCprint(X%(I%)*2-I%,Y%(I%)*2,233)
4080 ENDPROC

4500 DEF PROCblank(I%,X%,Y%)
4510 IF Q%(I%)>127 AND Q%(I%)<132 THEN COLOUR 2
4520 IF Q%(I%)>131 AND Q%(I%)<137 THEN COLOUR 3
4525 IF Q%(I%)=137 THEN COLOUR 3-I%
4530 PROCprint(X%,Y%,Q%(I%))
4540 ENDPROC

5000 DEF PROCthrow(I%)
5010 IF DICE THEN PROCdice1 ELSE PROCdice2
5020 REPEAT
```

```
 5030 PROCone_move(I%)
 5035 SOUND 1,-15,80+I%*8,2
 5040 M%=M%-1
 5050 HOME%=(X%(I%)=1 AND Y%(I%)=1)
 5060 UNTIL M%=0 OR HOME
 5065 IF SNAKE AND LADDER THEN PROCchoose
 5070 IF SNAKE THEN PROCupdown(I%,+1,131)
 5080 IF LADDER THEN PROCupdown(I%,-1,132)
 5090 HOME=(X%(I%)=1 AND Y%(I%)=1)
 5100 ENDPROC

 5200 DEF PROCchoose
 5210 IF RND(1)>.5 THEN LADDER=FALSE ELSE
SNAKE=FALSE
 5220 ENDPROC

 5500 DEF PROCdice1
 5501 COLOUR 128:COLOUR 2+I%
 5505 PRINT TAB(0,27);SPC(60);
 5510 PRINT TAB(0,27);"It's your throw"
 5520 PRINT " -";N$(I%)
 5530 INPUT "What did you get ",M%
 5540 IF M%<1 OR M%>6 THEN GOTO 5510
 5550 ENDPROC

 5600 DEF PROCdice2
 5601 COLOUR 128:COLOUR 2+I%
 5610 PRINT TAB(0,28);SPC(60);
 5620 PRINT TAB(0,28);"It's your throw"
 5630 PRINT " -";N$(I%)
 5640 PRINT "PRESS ANY KEY"
 5645 *FX 15,0
 5650 REPEAT
 5660 M%=RND(6)
 5670 PRINT TAB(0,31);M%;
 5680 UNTIL INKEY(0)<>-1
 5690 ENDPROC

 6000 DEF PROCupdown(I%,D%,C%)
 6010 LOCAL X%,Y%
 6020 PROCblank(I%,X%(I%)*2-I%,Y%(I%)*2)
 6030 X%=X%(I%)*2:Y%=Y%(I%)*2
 6040 REPEAT
 6050 Q%(I%)=FNchar(X%,Y%)
 6055 COLOUR 2+I%
```

```
6060 PROCprint(X%,Y%,233)
6065 SOUND 1,-15,50-2*Y%,2
6070 PROCblank(I%,X%,Y%)
6080 X%=X%-1:Y%=Y%+D%
6090 UNTIL Q%(I%)=C% OR FNon_man
6095 X%=X%+1:Y%=Y%-D%
6100 PROCblank(I%,X%,Y%)
6110 X%(I%)=X% DIV 2
6120 Y%(I%)=Y% DIV 2
6125 IF D%=1 THEN Y%(I%)=Y%(I%)+1:
     X%(I%)=X%(I%)+1
6130 Q%(I%)=FNchar(X%(I%)*2-I%,Y%(I%)*2)
6135 COLOUR 2+I%
6140 PROCprint(X%(I%)*2-I%,Y%(I%)*2,233)
6150 IF (Y%(I%) DIV 2)*2=Y%(I%) THEN
     VX%(I%)=1 ELSE VX%(I%)=-1
6160 ENDPROC

6500 DEF FNon_man
6510 LOCAL J%
6515 IF Q%(I%)<>137 THEN =FALSE
6520 IF I%=1 THEN J%=0 ELSE J%=1
6530 IF Q%(J%)=132 THEN =TRUE ELSE=FALSE

8000 DEF PROCtitle
8010 COLOUR 128+0
8020 COLOUR 1
8025 CLS
8030 PRINT TAB(3,3)"S N A K E S  A N D
     L A D D E R S"
8040 PRINT TAB(0,10);
8050 PRINT TAB(3);"This is a game for two
     players"
8060 PRINT TAB(2);"The first player to reach
     the top"
8070 PRINT TAB(2);"left of the board wins!!"
8080 PRINT TAB(0,20);
8090 INPUT "Do you want to use a real dice",A$
8100 A$=LEFT$(A$,1)
8110 IF A$<>"Y" AND A$<>"N" THEN GOTO 8010
8120 IF A$="Y" THEN DICE=TRUE ELSE DICE=FALSE
8140 PRINT TAB(0,25);
8150 INPUT "What is the first players
     name",N$(0)
8155 PRINT TAB(0,27)
```

```
8160 INPUT "What is the second players
     name",N$(1)
8170 ENDPROC

8500 DEF PROCendgame
8505 *FX 15,0
8510 COLOUR 128:COLOUR 2+I%
8520 PRINT TAB(0,27);SPC(99);
8525 PRINT TAB(0,27);
8530 PRINT "You win ";N$(I%)
8535 PRINT TAB(0,28);
8540 INPUT "Another game ",A$
8550 A$=LEFT$(A$,1)
8560 IF A$<>"Y" AND A$<>"N" THEN GOTO 8535
8570 IF A$="Y" THEN AGAIN=TRUE ELSE AGAIN=FALSE
8580 ENDPROC

9000 DEF FNclash(X%,Y%,L%,D%)
9005 LOCAL T%
9006 T%=FALSE
9007 E%=2*(X%+D%*L%)+D%:X%=X%*2-D%:Y%=Y%*2-1
9010 REPEAT
9020 IF FNchar(X%,Y%)<>32 THEN T%=TRUE
9025 IF FNchar(X%+1,Y%)<>32 THEN T%=TRUE
9026 IF FNchar(X%-1,Y%)<>32 THEN T%=TRUE
9030 X%=X%+D%:Y%=Y%+1
9040 UNTIL X%=E% OR T%=TRUE
9050 =T%

9500 DEF FNC(X%,Y%)
9510 X%=8+64*X%
9520 Y%=1023-32*Y%
9530 =POINT(X%,Y%)

9600 DEF PROCprint(X%,Y%,C%)
9610 COLOUR 128+FNC(X%,Y%)
9620 PRINT TAB(X%,Y%);CHR$(C%);
9630 ENDPROC

9700 DEF FNchar(X%,Y%)
9710 LOCAL A%
9715 COLOUR 128+FNC(X%,Y%)
9720 VDU 31,X%,Y%
9730 A%=135
9740 =(USR(&FFF4) AND &FF00) DIV &100
```

Action Snakes and Ladders

The basic form of Snakes and Ladders lends itself to a conversion to a dynamic graphics game. The idea is that instead of alternately moving each man by a random amount, a single man is continuously animated, moving one square each time through the animation loop. Of course, in this scheme the man will automatically slide down every snake and climb up every ladder along his path. The arrangement of ladders and snakes is such that the man sometimes reaches the final position and Sometimes is kept circling round, going up the same ladders and down the same snakes. To turn this into a game of skill all that is necessary is to add the extra condition that for the man to climb a ladder the player has to press the 'L' key just when the man reaches its foot and to avoid going down a snake the player has to press the 'S' key just as the man reaches the snake's head. Holding down either key at any other time causes the man to stop moving and, as the object of the game is to get the man to the top right hand corner as quickly as possible, this is not a good way to play the game!

The modifications to the Snakes and Ladders program are fairly simple. The main program becomes:

```
 10 REM Action Snakes and Ladders
 20 MODE 4
 30 PROCinit
 40 PROCtitle
 50 MODE 5:PROCcolour
 60 PROCp_sandl
 70 PROCstart
 80 REPEAT
100 PROCthrow(1)
110 UNTIL HOME
120 PROCendgame
130 IF AGAIN=TRUE THEN RUN
140 END
```

PROCstart has to be modified to print only one man character and to zero the time. This involves changing line 3710 and adding line 3755:

```
3710 FOR I%=1 TO 1

3755 TIME=0
```

The modifications and additions to PROCthrow are sufficient to make it worth giving a complete listing of the new version of the procedure:

```
5000 DEF PROCthrow(I%)
5010 PROCp_time
5030 PROCone_move(I%)
5035 SOUND 1,-15,80+I%*8,2
5040 M%=M%-1
5050 HOME=(X%(I%)=1 AND Y%(I%)=1)
5065 IF SNAKE AND LADDER THEN PROCchoose
5066 IF NOT SNAKE AND NOT LADDER AND (INKEY(-82)
     OR INKEY(-87)) THEN PROCp_time:GOTO 5066
5070 IF SNAKE AND NOT(INKEY(-82)) THEN
     PROCupdown(I%,+1,131)
5080 IF LADDER AND INKEY(-87) THEN
     PROCupdown(I%,-1,132)
5090 HOME=(X%(I%)=1 AND Y%(I%)=1)
5100 ENDPROC
```

Now PROCthrow only calls PROCone_move once each time through the animation loop and calls PROCp_time, a new but very short procedure that prints the current time at the bottom of the screen.

```
 5300 DEF PROCp_time
 5310 PRINT TAB(2,28);"Time=";TIME/100
 5320 ENDPROC
```

The only other modifications are the addition of line 5066, which stops the animation if either L or S is pressed and the man is not on the head of a snake or at the foot of a ladder, and the changes to lines 5070 and 5080 which result in the man sliding down a snake if S is not pressed and climbing a ladder only if L is pressed.

The changes to PROCtitle are simple but extensive and this makes it worth listing the entire new version of the procedure. The same is true of PROCendgame:

```
 8000 DEF PROCtitle
 8010 COLOUR 128+0
 8020 COLOUR 1
 8025 CLS
 8030 PRINT TAB(3,3)"S N A K E S  A N D  L A D D
E R S"
 8040 PRINT TAB(0,10);
 8050 PRINT TAB(2);"You must try to get your
      man to"
 8060 PRINT TAB(2);"the top lefthand corner of"
 8065 PRINT TAB(2);"the board as quickly as
      possible"
 8070 PRINT
```

```
8090 PRINT TAB(2);"You must avoid snakes
     heads by"
8100 PRINT TAB(2);"pressing S and you can climb"
8110 PRINT TAB(2);"ladders by pressing L"
8155 PRINT TAB(0,27)
8160 PRINT
8165 PRINT TAB(6,20);"Press any key to start"
8166 IF INKEY(0)=-1 THEN GOTO 8166
8170 ENDPROC

8500 DEF PROCendgame
8505 *FX 15,0
8510 COLOUR 128:COLOUR 3
8520 PRINT TAB(0,27);SPC(99);
8525 PRINT TAB(0,27);
8530 PRINT "You took ";TIME/100
8535 PRINT TAB(0,28);
8540 INPUT "Another game ",A$
8550 A$=LEFT$(A$,1)
8560 IF A$<>"Y" AND A$<>"N" THEN GOTO 8535
8570 IF A$="Y" THEN AGAIN=TRUE ELSE AGAIN=FALSE
8580 ENDPROC
```

With these changes to the original program you will find that you have a fast and very exciting game! The man moves around the board, down the snakes and up the ladders in a fascinating way. There are plenty of improvements that can be made to this fast Snakes and Ladders game along the lines of the features included in earlier games. For example, the end game routine could issue appropriate messages, you could impose a time limit, produce a table of best times, improve the sound effects . . . You could even introduce new elements to the game such as ‘black hole’ squares that have to be jumped over, etc. The lesson to be learned is that there is still a great deal that can be done with traditional games to bring them up to date and sometimes the results can be very impressive.

Chapter Eight
Becoming a Master Programmer

Graduating from being a novice programmer to being a master of the craft is a matter of developing the skills you have already acquired and gaining confidence to apply them to more and more ambitious projects. If you have typed in and used the programs in the earlier chapters then you should be beginning to see some of the ways in which large programs are written. However, to really get the most from what you have learned it is essential that you gain practical experience, first by modifying and extending the games given in this book and then by implementing your own ideas. You cannot become a good programmer without making the effort to write some programs and learn from your own mistakes and successes.

In this final chapter we look at some of the methods and attitudes that you should try to bring to bear on your work. Chapter One stressed the need to use a programming method and all the programs in this book have used a form of structured programming combined with stepwise refinement (a method described at greater length in Advanced Programming for the Electron (Mike James, Granada, 1984) but this is just part of the story. You could say that structured programming and stepwise refinement are the 'scientific side' ofthe craft of programming. Without their application, programming is hard work. H owever, even with the use of a programming method there is still a great deal of art left in good programming and this can only be learned by practice.

This is not to say that good programming is 'crafty' programming in the sense of relying on tricks and clever methods. A good program must be as clear as possible and tricks and clever methods tend to increase the confusion within a program and a confused program is a bug-prone program. As already mentioned the only route to becoming a master programmer is to learn from your own experiences of writing programs, but to do this you have first to know what to value and you have to finish the program that you start.

Games and problem solving

If you look back to earlier chapters you w ill find that most of the programs started as a simple idea which, implemented in BASIC, did not produce a very satisfactory game! The truth is that most ideas for games prove to be disappointing when first implemented but this is not a reason for abandoning the idea. The success of any computer game depends on how well it is implemented and even great ideas which are badly implemented will result in terrible games. For example, one of the oldest and most addictive computer games is 'Space Invaders', but it is possible to make even this game totally unplayable by not paying attention to detail. If the game runs too slowly or if the response to keypresses is sluggish then the game will be frustrating rather than exciting.

If a game doesn't live up to your expectations then do notimmediately discard the idea and look for something new; ask yourself the question what is wrong? Sometimes the answer will be simple to discover and not difficult to correct. For example, if the game runs too slowly then changing some of the routines into assembler should do the trick. On the other hand the reason for a sluggish keyboard response can often be difficult to track down. It may be due to the time that the animation loop takes to execute but it may just as easily be due to not Inspecting the keyboard for long enough each time through the loop. For example, if the animation loop runs too fast then the most obvious way to slow it down is to insert a delay loop. If the time spent in the delay loop is a significant part of the time the animation loop takes, then the keyboard will be examined for only a short period each time through the animation loop. The solution to this problem is not to speed up the animation loop but either to make use of the Electron's keyboard buffer (i.e. use lNKEY(0)) or to examine the keyboard more than once each time through the loop.

Sometimes the reason why the first attempt at a game is not successful doesn't lie in the implementation. For example, in Chapter Two the first version of Ant Hill was not as much fun to play as it should have been because it was too easy. If a game is too easy then the tendency is often to make it more difficult in the most obvious ways. For example, you can make any game more difficult by setting a short time limit but unless the time it takes to complete th game really does depend on the skill of the player, setting a short time limit either has no effect or makes the game impossible! What you have to do is examine the game carefully and discover what it is that makes it easy or

difficult. In the case of Ant Hill the problem was tracked down to the ants not posing enough of a problem to the man trying to get to the nest. In that particular case the solution was to make the dispersion of the ants at the start of the program more random and to confine them to a smaller area. If after examining the game the reason for it being too easy or too difficult turns out to be something that is virtually impossible to correct, then you have no choice but to give up and try something else but this is a very rare event.

The point is that very few games, or any other sort of program for that matter, are so easy that the first version that you produce fulfills your expectations and this is not a good reason to give up and start something else. If you want to improve your programming then you must make an effort to turn what you have in the way of a program into what you want and this is a process of tracking down and solving each of the small problems that keep your program from living up to its specifications.

Adding the finishing touches

Once you have a game that is fun to play, the next step is to add the procedures necessary to make it 'playable'. In other words you have to make a game fit to be used by other people and the only way you can discover if you have been successful is to watch other people play it. There is no point in claiming that players make your game crash because they misunderstood it and so the fault is theirs; if they misunderstood it, either you didn't tell them enough about it or it is too complicated and in any case a well-written game shouldn't crash!

All of the games in this book have been 'crash-proofed' in the sense that it should be impossible to input values or control the game in such a way as to make it stop working and drop the player back to BASIC. The main method of crash-proofing is based on the old and well known computer idea of Garbage In Garbage Out (often shortened to GIGO). If you can keep garbage from getting into your program you should be reasonably sure that it will not crash. For example, if you ask the player for a difficulty level then you should always check that it is in the correct range before moving on to the rest of the program. However, even if you check every input it is still possible for a program to crash or misbehave because of a combination of values that you haven't allowed for. For example, in the early stages of testing Snakes and Ladders a board was generated that included a square which contained the head of a snake ant! the tom of a ladder. The result was that the

man-shaped character both climbed the ladder and slid down the snake, with obvious and disastrous consequences for the rest of the program! A line had to be inserted using the procedure PROCchoose to make the man character choose only one of the two options.

To make sure that your program will not crash because of unexpected circumstances you must test it well. The theory of testing and debugging a program is covered in detail in Advanced Programmng for the Electron, but it is worth saying here that playing a game for any length of time is not the same as testing it! If you play a game as you are developing it then the chances are that by the time you reach a nearly finished version you Will be quite good at it. This means that you are testing the game at a particular level of skill and the problems that cause a crash could lurk at a much lower level of skill. For example, suppose you were testing an early version of Leap Frog (Chapter Three) then you might play many successul games only to discover that as soon as a beginner played the game it crashed because the player failed to bounce any of the ten frogs and so scored zero. When you test a game you must try out all the extreme possibilities. high scores, low scores and play it in the 'silliest' fashion you can think of in short, try to crash it. If you cannot make it crash when you are trying your hardest then With luck any problems it contains should only be small ones!

There is a particular problem with testing games programs that doesn't generally occur With other types of program and that's randomness. For example, the board printed as part of Snakes and Ladders is generated at random and you could wait a long time before a 'problem' board (i.e. the one containing a snake's head and the foot of a ladder in the same square) is generated. There are two ways of dealing with the problem of testing random sections of the program. First, you can change any statements that contain RND functions within the program to set the variables to fixed values that might cause problems. The trouble with this method is that you have to think of the values that are likely to cause trouble and in general, if you can do this, the problem is so obvious that you don't even need to run the program! For example, if you spot the fact that values of the random numbers in the procedures that print a snake and print a ladder in Snakes and Ladders can result in a snake's head and the foot of a ladder being printed in the same square, then it is obvious that this is going to cause a problem.

The second method of testing the random component of a game is simply to look at a great many examples. This is one of the methods that was used to test Snakes and Ladders during its development. To make it possible to see a great many examples in a reasonable amount

of time the main program was changed into a loop that just called the procedures that printed the board over and over again. Once a problem has been located in this way, it is important to make sure that it can he made to happen 'on request' before trying to fix the bug. In the case of Snakes and Ladders, once a board has been generated with a snake's head and the foot of a ladder in the same square, the RND functions were replaced by constants that produced this set up. That is, the program was modified so that every time it was run it produced a problem board. This is the only way that the effectiveness of a bug fix can be tested in a program that uses randomness. For example, imagine that after a few hours of testing a problem crops up that crashes the program; if the bug is fixed immediately you will then have to wait for another few hours to tell if your fix works or not. On the other hand, if you first find a way to modify the program so that the problem crops up each time you run it, you can be sure that your bug fix works without having to test it for hours!

Testing and debugging are certainly areas where a great deal of the craft or programming is to be found. However, producing 'user friendly' programs is just as important and just as skillful. In some senses it is the most skillful part of programming because it is almost impossible to say what makes a program user-friendly. The only way to discover what is good in this respect is to watch the way users react to your program. When you are designing your program you must try to think of the way in which it will be used and so find the 'natural' order for the program to work in. However, even if you produce a program that you find easy to use, it has to be remembered that you are not a typical user! Programmers tend to forget how difficult and confusing a computer keyboard and display are to non-programmers and they certainly know their own program better than anyone else. As a result something that seems easy or obvious to you can seem awkward and obscure to a non-programming user. Every programmer has had the experience of watching someone else use their program in such a clumsy way that it caused them to despair at the user's stupidity! This is of course the wrong attitude: users are always right and if they find your program difficult to use then you must treat this as a problem with your program, not with the user!

The range of games

If you are inspired to try your hand at inventing and writing your own games programs then you might be interested in knowing something about the types of games that are currently popular.

Arcade or animation games form the largest and most popular group of games. With the possible exception of Snakes and Ladders, all of the games this book are based on animation. The main feature of all such games is the way that the player has control of the motion of some object on the screen. Although you can create new animation games by thinking up fresh themes and objectives the only way in which the games really differ is in the way that the player can control the moving object. For example, you could make a new game out of Ant Hill by changing all the ants to ghosts and the nest to a treasure chest but the game would still play like Ant Hill. However, Snake in Chapter Five is very different from Ant Hill in the skill that the player needs to control the moving object. If you want to create a new animation game then you need to think not only of the theme of the game but also of how the player will control the moving object and what new element of skill is needed.

The second most popular group of games are 'adventure games'. These sometimes contain simple animation games within them. but only as part of the overall objective of the game. An adventure game doesn't rely on animation to test the player's skill; instead it creates a simulated world inhabited by a variety of creatures, some helpful, some hostile. The world and its creatures are most often created using descriptions and limited graphics and the player generally interacts with the world by commands typed on the keyboard. Adventure games generally present less of a programming problem than animation games but they do need a great deal of imagination.

The third category of games, games of strategy, include computer chess, noughts and crosses, draughts etc. Games of strategy can sometimes involve complicated graphics (consider the problems of printing a chess board complete with pieces, for example), but the main programming challenge they present is in making the computer play a convincing game. Working out how to program a computer to play a game like chess will take you into the frontiers of computer science.

When you are working out new games there is always the element of the surprise discovery. It is amazing how often implementing one game immediately suggests another that uses the same elements in different ways. This is yet another reason for always writing programs using procedures. If a game has been written using procedures, they can often be re-used in the construction of other games and in this sense the more games you write, the easier it gets.

Using the Electron

The Electron is an amazing machine but it does have its limitations. In fact it would be better to say that there are features of the machine that make it easier to use in some ways rather than others. For example, if you want to use a 40-column screen and four colours then you will have to be careful about how you use memory (there is very little left over after mode 1 takes its 20K!). What this means is that it is better to try to implement games either using two colours and 40 columns or four colours and 20 columns.

One of the biggest problems with using the Electron is that it is a little on the slow side. This is due to the way the hardware works and there is little that can be done about it. As a result it is better to avoid the use of modes that involve a large number of memory accesses to print a single character. Once again this implies that two colours and 40 columns or four colours and 20 columns are the easiest modes to use

You may be surprised that the excellent high resolution graphics capabilities of the Electron weren't used in any of the games in this book. This is fairly typical in that the main use of high resolution graphics in games is to create background. Most animation games are much easier to implement using user-defined graphics and the PRINT command.

The way ahead

If you have played with the games in this book in the sense of changing and generally tinkering with them then the next thing to do is to write a games program of your own. Y our aim should be to produce a program that other people can use and enjoy rather than something half finished that only you can play. If you do this then you will find that you benefit twice from the program once from the fun in playing the game and once from the satisfaction in having finished something that other people can play.