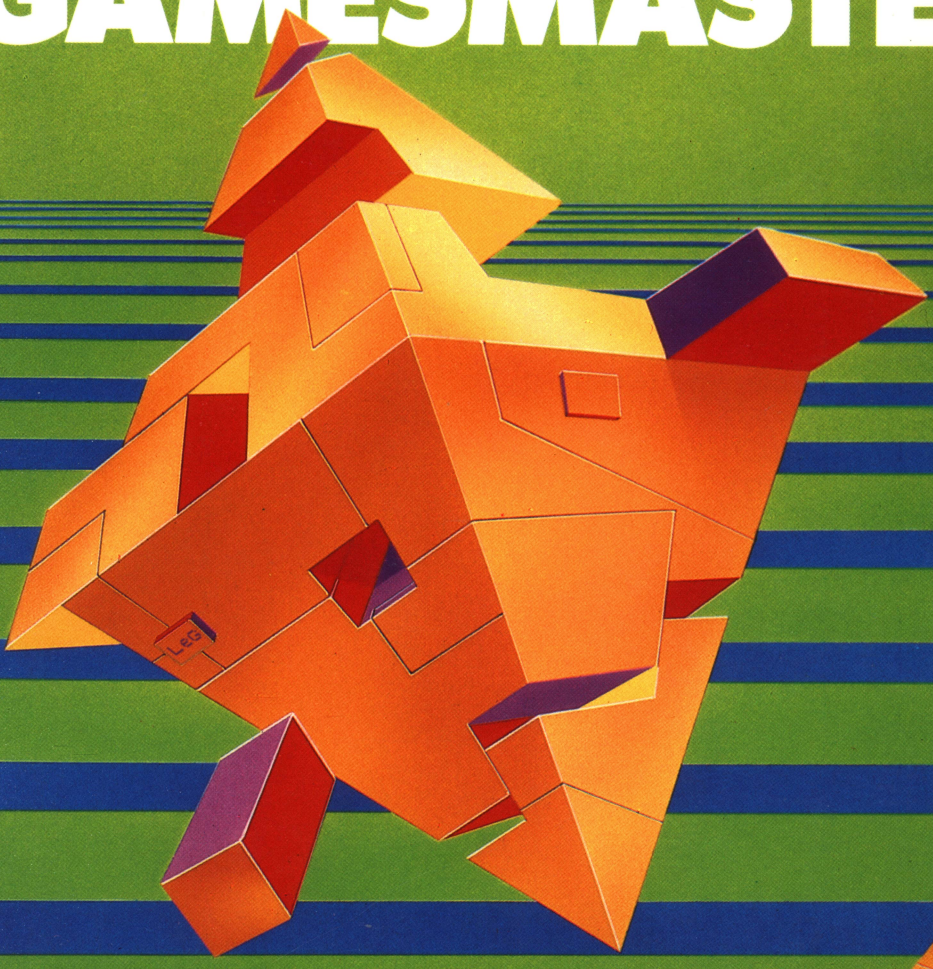
QL GAMESMASTER



KAY EWBANK,
MIKE JAMES
AND S M GEE

Dit boek is eigendom van:
Eric Nijenhuis
& Geert Jan Lootsma

The QL Gamesmaster

More books for QL users

QL SuperBASIC
A. A. Berk
0 246 12596 9

QL Computing
Ian Sinclair
0 246 12595 0

Professional and Business Uses of the QL
Colin Lewis
0 00 383005 5

The QL Gamesmaster

Kay Ewbank, Mike James and
S. M. Gee

COLLINS
8 Grafton Street, London W1

Collins Professional and Technical Books
William Collins Sons & Co. Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Collins Professional and Technical Books 1985

Distributed in the United States of America
by Sheridan House, Inc.

Copyright © Kay Ewbank, Mike James and S. M. Gee 1985

British Library Cataloguing in Publication Data
Ewbank, K.
The QL Gamesmaster
1. Computer games 2. Sinclair QL (Computer)
— Programming
I. Title II. James, M. III. Gee, S.M.
794.8'028'542 GV1469.2

ISBN 0-00-383066-7

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

All rights reserved. No part of this publication may
be reproduced, stored in a retrieval system or transmitted,
in any form, or by any means, electronic, mechanical, photocopying,
recording or otherwise, without the prior permission of the
publishers.

Contents

Preface ix

1. The Sorcerer's Apprentice 1
   - How to write programs 2
   - The need for method 4
   - Stepwise refinement and procedures 4
   - Structure and style 5
   - A games mode for the QL 7
   - Procedures *def_graph* and *graph* 10
   - Finding out the colour – *pix* 12
   - Games techniques 13
2. Ant Hill 15
   - The game design 15
   - The technique of animation 18
   - The main program 20
   - Procedures *init_graph, init* and *title* 21
   - Procedure *draw_scene* 25
   - Procedure *one_move,* – the heart of the game 27
   - Procedure *end_game* 31
   - Making the game better 35
   - Suggestions for further work 37
   - The final version – a complete listing 37
3. Leap Frog 47
   - The game design 47
   - Bouncing and falling 48

The main program 50
Procedures *init_graph, init* and *title* 51
Procedure *print_flies* 54
Procedure *startup* 55
Procedures *move_ss, print_ss* and *move_f* 56
Procedures *seesaw, hit_fly* and *miss* 57
Procedure *end_game* 59
Evaluation and improvements 60
The final version – a complete listing 61
4. Frogling 68
Scrolling animation 68
The game design 69
The use of text windows 70
Main program and procedure *title* 71
Procedures *init_graph*, init and *get_time* 73
Procedure *scrll* and its associated procedures 76
Procedures *move_f, change_char* and *off_s* 79
Procedures *death* and *end_game* 81
Evaluation and improvements 82
The final version – a complete listing 83
5. Snake 90
Snake animation 90
The directional snake – the queue 92
The game design 95
The main program 96
Procedures *init_graph* and *init* 97
The title frame 100
Procedures *draw_scene, draw_snake* and *draw_food* 102
Procedure *one_move* and its associated procedures 104
Procedure *hit_f* and its associated procedures 106
Procedure *end_game* 108
Conclusion – playing the game 111

The final version – a complete listing 111
6. Tadpole 120
The game design 120
Setting the scene 121
Making an escape 123
Catching frogs 125
Conclusion 127
The final program – a complete listing 127
7. Snakes and Ladders 138
The game design 138
Main program 140
Procedures *init_graph* and *title* 141
Printing the board – *p_sandl* and its associated procedures 145
Procedure *start* 149
Moving men – *throw* and its associated procedures 150
Procedure *end_game* 155
Evaluation 156
The complete listing 156
Action Snakes and Ladders 165
8. Becoming a Master Programmer 169
Games and problem solving 170
Finishing 171
The range of games 173
Using the QL 175
The way ahead 175

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. Lots of 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.

Given that our aim is to teach the art of writing large programs, you may be wondering why we have chosen to concentrate on games. There are two good reasons. For one thing games are fun – fun to play and also fun to design. More important from the teaching point of view is the fact that they combine all the aspects of programming; where else would you find graphics and sound effects put together with random numbers and Boolean logic? The programs developed in the book are all animated graphics games and some of them at least are familiar classic arcade games. You may have thought that programming such games was beyond your reach but this book shows you how to go about writing games even of this level of sophistication and complexity.

Before embarking on the design of the individual games, we had to find a remedy for the QL's lack of built-in block graphics, which are essential to the type of games we wanted to implement. The solution we decided on was to add a completely new graphics mode – a low resolution mode which complements the two high resolution modes already available on the QL. Extending the QL's facilities in this way proved much more straightforward than you might have imagined and simply required two new procedures which are explained in Chapter One and then used throughout the rest of the book.

All the games 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 it 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 Collins without whose help this book would not have been possible.

Kay Ewbank
Mike James
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 QL 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* anything 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 can 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 to 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 also a pleasure to use.

The QL 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. If you want simply to use the book as a collection of games then there is nothing stopping you – just turn to the complete listing given at the end of each chapter and type it 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 QL 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 each section of the program as it 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.

At 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 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 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 programmer had, or simply to leave them as complex tangles that might be usable but are impossible to understand or modify. The programs in *The QL Gamesmaster* haven't been polished at all and 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 QL 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.

Though this is a practical book, even the most practical subjects need some theory and so the next part of this chapter describes some of the ideas that lie behind programming.

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 situtation is a great pity because it represents many hours of wasted programming effort. The truth is that if you use a programming *method* 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 writing 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 down into a collection of smaller programs. The smaller programs can then be divided down 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 to 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 in 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 SuperBASIC:

```
10 draw_board
20 REPeat game
30 human_move
40 machine_move
50 IF checkmate THEN EXIT game
60 END REPeat game
70 STOP
```

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 would be written by dividing them down into a list of even smaller procedure calls and so on until the program was written. Notice that by 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 refinement 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 of using stepwise refinement and procedures that it is impossible to list them all. However, many of the advantages will become apparent from the examples in the following chapters.

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 a procedure has to do something other than just call more 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 GOTO too freely, then you will discover that your programs are impossible to understand and debug. It is very difficult to 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 is 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 *print_board* rather than *xy4*. If you can think up appropriate names for variables and procedures 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 – and 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 at or (like most programmers) bad at, but it is definitely worth the effort of trying.

All this talk of program clarity and style may be worrying if you

have always been taught that the most 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 every variable. However, if you produce a fast program with plenty of memory left over for additions and modifications and yet no-one can understand it (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 68008 used in the QL is a very good microprocessor to write assembly language for. Unfortunately, at the moment there is no standard 68008 assembler available and this makes it difficult to include assembler within any of the games in this book. However, assembly language isn't totally ignored, as you will discover in the next section.

A games mode for the QL

The QL's two high resolution graphics modes are excellent for drawing graphs and a wide range of other applications but they are not really suitable for games programs. The reason for this is that the high resolution commands work in terms of lines and points, and games programs involve moving solid shapes around the screen. The obvious solution is to use text mode commands, i.e. AT and PRINT in combination with user-defined graphics. However, the QL's user-defined graphics (described in an information sheet available from Sinclair Research) are only 5 by 9 pixels in size and each is surrounded by a border of background points. As a result it is difficult to define characters with sufficient resolution to be interesting and it is impossible to join characters together to form a larger continuous shape.

The solution to the user-defined graphics problem which is adopted in this book – to add a completely new graphics mode – may seem a little drastic but in fact it is not difficult and serves to prove how easy it is to extend the QL's facilities. The new graphics mode

converts mode 8 (256 by 256 pixels) into a low resolution graphics mode using 8 by 8 pixel graphics characters in 32 lines of 32 characters. What is surprising is that creating this new graphics mode requires only two procedures – *def_graph*, used to define graphics characters and *graph*, used to print graphics characters on the screen. The basic idea is that *def_graph* stores the pattern of bits needed to define the shape and colour of a graphics character in a table, and *graph* simply transfers these from the table to the correct position on the screen. The syntax of *def_graph* is:

```
def_graph s,f,b,r0,r1,r2,r3,r4,r5,r6,r7
```

where s is the number of the graphics character being defined, f is the foreground colour, b is the background colour and r0 to r7 are numbers that define the pattern of each row of dots in the character. Notice that in this definition the foreground and background colour of the character is fixed and cannot be changed by a later use of INK or PAPER. This has the advantage of increasing the speed with which a character can be transferred to the screen and the disadvantage that if the same shape is required with a number of different foreground-background combinations then a new graphics character has to be used for each.

The way in which the numbers r0 to r7 define the pattern of a row of dots will be familiar to anyone who has used another personal computer, for example the Spectrum. If you write down the pattern of eight dots using 0 to stand for the background dots and 1 to stand for the foreground dots then the number that represents the entire pattern is obtained by multiplying each of the ones or zeros by an appropriate 'weight' and adding up all the results. The weights for each dot position are:

leftmost dots rightmost dots

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

so, for example, the dot pattern:

*___****

(where * is a foreground dot and _ is a background dot) can be written as

10001111

which, when each zero or one is multiplied by the appropriate weight and added up, gives:

128×1+64×0+32×0+16×0+8×1+4×1+2×1+1×1

or, after working everything out, 143. Of course, to define an entire graphics character this process has to be repeated for each of the eight rows of dots. For example, to define two graphics characters, a man and a solid block, two calls have to be made to *def_graph*:

```
def_graph 0,7,0,24,24,126,24,60,102,195,195
def_graph 1,2,0,255,255,255,255,255,255,255,255
```

The first call defines graphics character zero to be a white man shape against a black background and the second defines graphics character one to be a red solid block.

Once a graphics character has been defined it can be placed anywhere on the 32 by 32 screen by using:

```
graph x,y,s
```

where x and y are the horizontal and vertical positions respectively and s is the number of the graphics character to be printed. The character rows are numbered from the top of the screen starting from zero and similarly the character columns are numbered from the left starting from zero. So:

```
graph 0,0,1
```

will print graphics character 1 in the top left-hand corner and

```
graph 31,31,1
```

will print graphics character 1 in the bottom right-hand corner.

As well as using *def_graph* and *graph* it is also necessary to reserve memory for the table of graphics character definitions. However, this is not difficult: if a program is using N graphics characters then as each definition requires 32 bytes to store, the line

```
C_TAB=RESPR(32*N)
```

should be included. (C_TAB is simply the name given to the character table used by *def_graph* and *graph*.)

These two procedures are really all there is to using the new graphics mode. Of course, you have to remember to include *def_graph* and *graph* in the program, but apart from this the new low resolution mode is as easy to use as the built-in high resolution modes. The next section gives a listing and brief explanation of both *def_graph* and *graph*. If you are not interested in how they work then simply skip the descriptions but do not forget to include the procedures in the programs in the following chapters.

Procedures *def_graph* and *graph*

The format of the data that has to be stored in memory to produce any given pattern of coloured dots is explained in the *QL User Guide*. Essentially, the colour of a group of four pixels is controlled by the sixteen bits in a single word (i.e. two bytes) as shown below:

pixel	0		1		2		3		0		1		2		3	
	G	F	G	F	G	F	G	F	R	B	R	B	R	B	R	B
bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

where G=green F=flash, R=red and B=blue

The colour of each pixel is determined by four bits, for example pixel 1 is controlled by bits 13, 12, 5 and 4, which give the state of the green, flash, red and blue component of the video signal respectively. Using this and the information that the screen memory starts at address 131072 (corresponding to the four pixels at the top left-hand corner of the display) and continues on in line order, it is fairly easy to write *def_graph* using nothing but SuperBASIC. However, it does involve rather a lot of 'bit manipulation' so the resulting procedure is a little difficult to understand:

```
1300 DEFine PROCedure def_graph(S%,F%,B%,
     R0%,R1%,R2%,R3%,R4%,R5%,R6%,R7%)
1310 LOCal A,FH%,FL%,BH%,BL%
1320 A=C_TAB+32*S%
1330 FH%=(F% DIV 4)*2
1340 FL%=F% && 3
1350 BH%=(B% DIV 4)*2
1360 BL%=B% && 3
1370 charline(R0%)
1380 charline(R1%)
1390 charline(R2%)
1400 charline(R3%)
1410 charline(R4%)
1420 charline(R5%)
1430 charline(R6%)
1440 charline(R7%)
1499 END DEFine def_graph
1500 DEFine PROCedure charline(R%)
1510 LOCal M%,J,I,DL%,DH%
1520 M%=128
1530 FOR J=1 TO 2
1540  DL%=0:DH%=0
1550  FOR I=1 TO 4
1560   IF (R% && M%)=M% THEN
1570    DL%=FH%+DL%*4
```

```
1580     DH%=FL%+DH%*4
1590    ELSE
1600     DL%=BH%+DL%*4
1610     DH%=BL%+DH%*4
1620    END IF
1630    M%=M% DIV 2
1640   END FOR I
1650   POKE A,DL%
1660   POKE A+1,DH%
1670   A=A+2
1680 END FOR J
1699 END DEFine charline
```

Procedure *def_graph* first makes up the bit patterns corresponding to the foreground and background colours (lines 1330 to 1360) and then calls procedure *charline* to store the bit pattern corresponding to each row of the character definition in the character table.

Procedure *graph* is a lot simpler than *def_graph*:

```
1700 DEFine PROCedure graph(X%,Y%,S%)
1710 LOCal A,B,I
1720 A=C_TAB+32*S%
1730 B=131072+(X%+Y%*256)*4
1740 FOR I=0 TO 7
1750  POKE_L B,PEEK_L(A+I*4)
1760  B=B+128
1770 END FOR I
1799 END DEFine graph
```

Line 1720 calculates the starting address of the character table, line 1730 calculates the address of the first screen memory location within the character location specified by X% and Y%, and finally line 1750 transfers the bit pattern from the character table to the screen. This is repeated within the FOR loop (lines 1740 to 1770) for each of the eight pixel rows of the character.

This version of *graph* does indeed work, but for most games programs it is far too slow. The only solution to this problem is to resort to 68000 machine code. Converting *graph* to machine code is not difficult, simply tedious:

```
 0    -5245      ASL #5,D3           E9 83
 2    -10109     ADD D3,D4           D8 83
 4    -7806      ASL #8,D2           E1 82
 6    -11646     ADD D2,D1           D2 80
 8    -6783      ASL #2,D1           E5 81
10    1665       ADDI #131072,D1     06 81
12    2                              00 02
14    0                              00 00
```

```
16    8769        MOVEA D1,A1          22 41
18    8316        MOVEA #0,A0          20 7C
20    0                                00 00
22    0                                00 00
24    9264 LOOP   MOVE 0(A0,D4),D2     24 30
26    18432                            48 00
28    8834        MOVE D2,(A1)         22 82
30   -11524       ADDA #128,A1         D2 FC
32    128                              00 80
34    22664       ADDQ #4,A0           58 88
36   -20228       CMPA #32,A0          B0 FC
38    32                               00 20
40    26350       BNE LOOP             66 EE
42    28672       MOVEQ 0,D0           70 00
44    20085       RTS                  4E 75
```

Using this machine code, procedure *graph* is nothing more than a CALL statement, but of course the machine code has to be loaded into memory somewhere within the program. The new version of *graph* is:

```
1700 DEFine PROCedure graph(X%,Y%,S%)
1710 CALL do_graph,X%,Y%,S%,C_TAB
1799 END DEFine graph
```

and a suitable machine code loader (including the machine code) is:

```
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430 READ B
440 POKE_W I,B
450 END FOR I
```

It is important to notice that procedure *def_graph* and the machine code version of *graph* are used in all the programs in this book and while they are included in the final listing at the end of each chapter they are not always re-listed as the program is developed.

Finding out the colour – *pix*

An important component of many games programs is the ability to find out what colour any given pixel currently is. Unfortunately, SuperBASIC lacks such a function, but once again this deficiency is

easy to rectify. Using a knowledge of the screen layout it is possible to write a function that will return the colour of any given pixel:

```
9500 DEFine FuNction pix(R%,C%,X%,Y%)
9510 LOCal A%,P%,B%,A
9520 P%=C%*8+X%
9530 A=131072+(512*R%+Y%*64+P% DIV 4)*2
9540 B%=PEEK(A+1)
9550 A%=PEEK(A)
9560 P%=P% MOD 4
9570 IF P%=3 THEN RETurn ((2 && A%)*2)+
     (3 && B%)
9580 IF P%=2 THEN RETurn ((8 && A%)DIV 2)+
     ((12 && B%) DIV 4)
9590 IF P%=1 THEN RETurn ((32 && A%)DIV 8)+
     ((48 && B%) DIV 16)
9600 IF P%=0 THEN RETurn ((128 && A%) DIV 32)+
     ((192 && B%) DIV 64)
9699 END DEFine pix
```

Function *pix* will return the colour of the pixel in the X% column and the Y% row of the character within row R% and column C%. In other words there are two stages involved in specifying the pixel to be examined – first, the character that contains it and second, its location within the character. So for example:

```
pix(10,11,5,4)
```

will return the colour code of a pixel within the character in the tenth row and eleventh column. The exact pixel examined is the one in the fifth column and fourth row of the character concerned. Function *pix* is used in all the programs in this book and it is important to note that while it is included in the final listing of each program, it is not repeated within the chapter.

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 called 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 QL 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. The major problem in implementing a game of this sort is in animating a large 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 not usually 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 naturally into three parts:

1. the background graphics,

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

The background for 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 possible route 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 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.

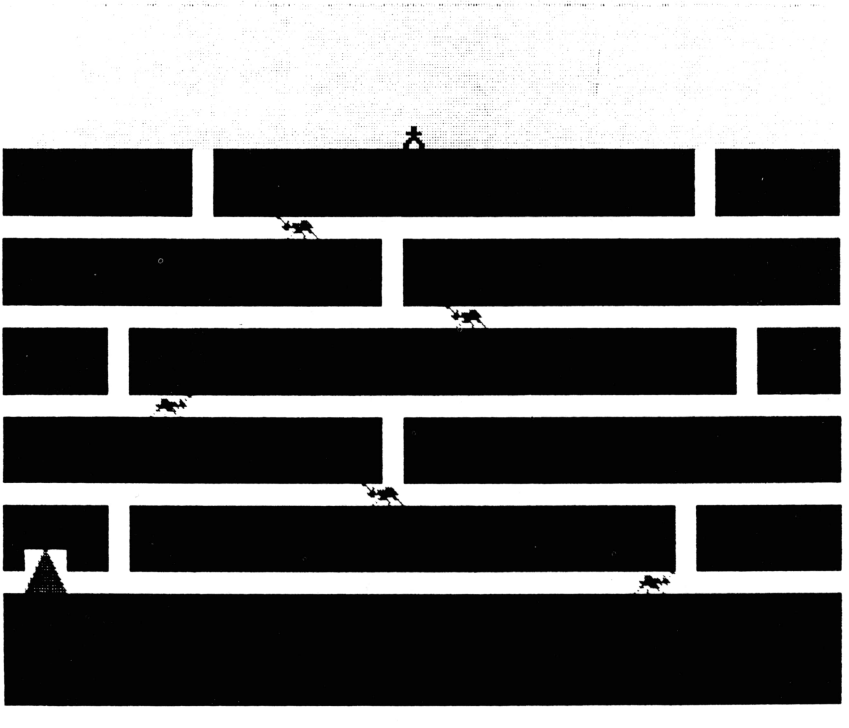


Fig. 2.1.

The man shape is easily implemented 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 have to be one character wide. However, the ant shapes are much more difficult to

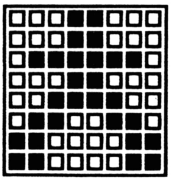


Fig. 2.2. Graphics character for man.

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.

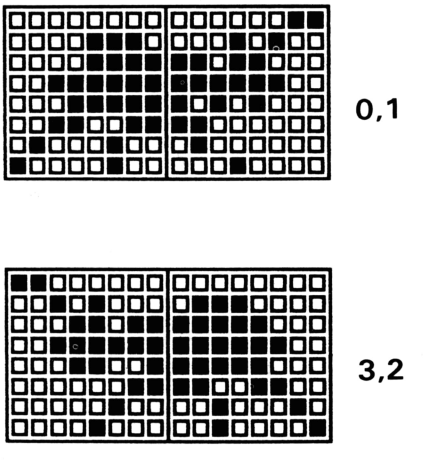


Fig. 2.3. Graphics characters for ants.

The rules of movement for the ants are easy to define. Each level will have one ant confined to that level and able to move along the tunnel. If an ant happens to arrive at the current location 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 sufficient 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 were also allowed to move back up the shafts so the player's control over the man is limited to the left and right arrow keys for horizontal motion and the down arrow key for vertical motion. 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 key 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, or
3. the time limit is exceeded and the eggs hatch.

Although there is no doubt that a successful game has to be exciting to play, 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 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 to print it repeatedly at one location, then erase it (usually by printing a blank) and then reprint it at its new location. If you can repeat the 'print-erase-print' cycle sufficiently fast then you can create the illusion of fairly smooth motion.

In practice, there are two factors that control how smooth computer-generated motion appears. The first is the time 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 we 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 also needs to be short but this isn't always so. The way that the different time intervals affect the animation can be seen in Fig. 2.4.

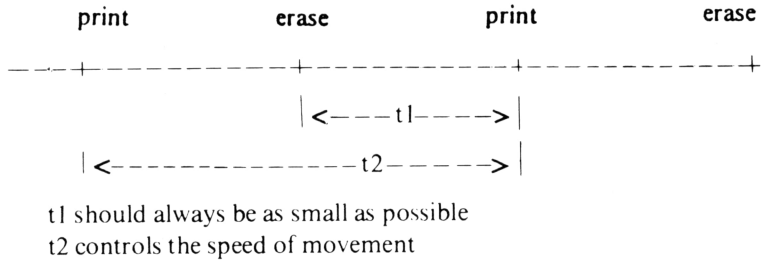


Fig. 2.4. Animation timing.

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 has 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 its X and Y screen co-ordinates; and the amounts that its X and Y co-ordinates should change each time through the animation 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 may be defined as graphics character S, its position in X and Y and its velocity in VX and VY. Its animation loop would be:

```
graph X,Y,B                    :REM  erase object
X=X+VX:Y=Y+VY                  :REM  update co-ordinates
graph X,Y,S                    :REM  reprint object
```

where graphics character B is a blank of the correct background colour (see Chapter One). In practice animation loops are generally a little more complicated because the object's velocity is usally changed as a consequence of where it is on the screen. For example, an animation of a bouncing ball would involve changing the velocity whenever the ball bounced against the screen boundary.

This 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 REMark ANT HILL
 20 MODE 8
 25 WINDOW 512,256,0,0
 26 title
 30 init_graph
 35 REPeat game
 36  init
 50  draw_scene
 60  REPeat main
 70   one_move
 80   IF TIME_UP OR HIT OR HOME THEN EXIT main
 90  END REPeat main
100  end_game
110  IF AGAIN=FALSE THEN EXIT game
120 END REPeat game
130 INK 0
140 PAPER 7
150 CLS
199 STOP
```

Lines 26 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

199 deal with the end of the game. Procedure *title* simply prints a title frame and sets the difficulty level of the game. Procedure *init_graph* sets up the machine code used by procedure *graph* (see Chapter One) and the graphics characters used by the rest of the program. Procedure *init* simply dimensions arrays and generally initialises any variables that need initialising. Notice that *init_graph* is only carried out once when the program is RUN but *init* is executed each time the player requests another game. Procedure *draw_scene* is intended to draw the system of tunnels and shafts ready for the game to start.

You could say that the heart of the program is procedure *one_move* which is responsible for making one move of both the man and the ants. Procedure *one_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 procedure *one_move*. Note that any numeric variables can be used to store either of the two 'truth values' TRUE or FALSE (set within *init_graph*) and so can be used by procedures to 'pass back' the result of a test. For example, procedure *end_game* not only supplies an appropriate end to the game, but also asks the player if another game is required and returns the result in AGAIN, using TRUE to signal yes and FALSE to signal no. All that now remains is to write each of the procedures used by the main program.

Procedures *init_graph, init* and *title*

The first three procedures, *init_graph, init* and *title*, were not difficult to write. Procedure *init_graph* and *init*, however, changed during the course of writing the rest of the program. 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 early versions of these procedures and then introduce all the changes as the other procedures are explained, it is less confusing to simply give them in their final form.

```
350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
```

```
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430 READ B
440 POKE_W I,B
450 END FOR I
520 C_TAB=RESPR(32*12)
530 def_graph 0,3,0,0,14,15,63,31,51,68,132
540 def_graph 1,3,0,3,20,216,252,168,192,64,16
550 def_graph 2,3,0,0,112,248,252,248,204,34,33
560 def_graph 3,3,0,192,40,27,63,25,3,4,8
570 def_graph 4,6,0,24,24,126,24,60,102,195,195
580 def_graph 5,4,0,1,1,3,3,7,7,15,15
590 def_graph 6,4,0,31,31,63,63,127,127,255,255
600 def_graph 7,4,0,128,128,192,192,224,224,240,240
610 def_graph 8,4,0,248,248,252,252,254,254,255,255
620 def_graph 9,0,0,255,255,255,255,255,255,255,255
630 def_graph 10,1,1,255,255,255,255,255,255,255,255
640 def_graph 11,6,1,24,24,126,24,60,102,195,195
650 FALSE=0
660 TRUE=1
670 AGAIN=TRUE
699 END DEFine init_graph
```

The first part of the procedure (lines 360 to 450) stores the machine code used by *graph* in a reserved area of memory as described in Chapter One. The second part of the procedure creates the necessary user-defined characters. Line 520 reserves enough storage for twelve graphics characters. Lines 530 and 540 define the two graphics characters that go together to form an ant shape facing right and lines 550 and 560 do the same thing for an ant shape facing left. That is, following procedure *init_graph*,

```
graph X,Y,0
graph X+1,Y,1
```

will print an ant facing right at X,Y and

```
graph X,Y,3
graph X+1,Y,2
```

will print an ant facing left 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 procedure *move_ant*. Lines 570 and 640 define two man-shaped graphics characters. Graphics character 4 is a man on a black background and graphics character 11 is a man on a blue background. Obviously, graphics character 4 has to be used in the tunnels and character 11 has to be used at the start of the game when the man is above ground and hence against a blue background (the sky). Lines 580 to 610 define four graphics characters that fit together to make up a pyramid shape that represents the nest. That is:

```
graph X,Y-1,5:graph X+1,Y-1,7
graph X,Y,6:graph X+1,Y,8
```

will print the four characters that make up the nest with its bottom left-hand corner at X,Y. Finally, graphics character 9, defined in line 620, and graphics character 10, defined in line 630, are solid blanking characters. Character 9 is black and used to blank objects within the tunnels and character 10 is blue and is used to blank objects above ground. A summary of the graphics characters can be seen in Table 2.1.

Table 2.1 Graphics characters

Number	Shape	Foreground	Background
0	rear of right ant	magenta	black
1	front of right ant	magenta	black
2	rear of left ant	magenta	black
3	front of left ant	magenta	black
4	man	yellow	black
5	top left of nest	green	black
6	bottom left of nest	green	black
7	top right of nest	green	black
8	bottom right of nest	green	black
9	black space	black	black
10	blue	blue	blue
11	man	yellow	blue

Procedure *init* is concerned with creating and initialising variables used later in the program.

```
1000 DEFine PROCedure init
1050 RANDOMISE
1060 XM%=RND(0 TO 20)+10
```

```
1070 YM%=2
1080 DIM A%(4,3),A1%(4,2)
1090 FOR I=0 TO 4
1100  A%(I,1)=RND(0 TO 25)+5
1110  A%(I,2)=(I+1)*5+2
1120  A%(I,3)=1
1130  A1%(I,1)=0:A1%(I,2)=1
1140 END FOR I
1160 AGAIN=FALSE
1170 HIT=FALSE
1180 HOME=FALSE
1190 TIME_UP=FALSE
1299 END DEFine init
```

Lines 1060 and 1070 set the man's initial position in XM% and YM%. The man's vertical position is always fixed at 2 so that he starts the game above ground but his horizontal position is random. Lines 1080 to 1140 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%(I,1) gives the X co-ordinate of ant I, A%(I,2) gives the associated Y co-ordinate and A%(I,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 A1%(4,2). Thus, to print ant I (remember I is in the range 0 to 4) use:

```
graph A%(I,1),A%(I,2),A1%(I,1)
graph A%(I,1)+1,A%(I,2),A1%(I,2)
```

Line 1100 initialises each ant's horizontal position randomly in the range 5 to 30. Line 1110 initialises each ant's vertical position so that there is one ant per tunnel. That is, ant 0 is printed on line 7, and 1 on line 12 and ant 2 on line 17, and so on. All the initial ant directions are set to 1, making all the ants move to the right (line 1120) and so line 1130 stores a right-facing ant shape in A1%(I,1) and A1%(I,2). The final part of procedure *init* sets the variables that are used to indicate the end of the game to FALSE (lines 1160 to 1190).

Procedure *title* is simplicity itself! Indeed, writing this procedure simply involves finding a reasonable form of words to explain the game to a new player:

```
8000 DEFine PROCedure title
8010 BORDER 0:INK 6:PAPER 0:CLS
8020 AT 3,14 :PRINT "A N T  H I L L"
8040 AT 7,3 : PRINT "In this game you have a
     fixed "
```

```
8050 PRINT TO 2;"amount of time to destroy
     the ants"
8060 PRINT TO 2;"nest before the ants hatch!"\
8070 PRINT TO 2;"BEWARE: the soldier ants
     guarding the"
8080 PRINT TO 2;"tunnels will capture you and
     take you"
8090 PRINT TO 2;"to the nest to feed the young!"\
8100 PRINT TO 2;"Good Luck!"
8200 REPeat diff
8300  AT 18,5 :PRINT "Which difficulty level -"
8310  PRINT TO 3;"1. Expert, 2. Medium or
      3. Novice ";
8320  INPUT DF
8330  IF DF>0 AND DF<4 THEN EXIT diff
8335 END REPeat diff
8340 MAX=(20+DF*10)
8350 AT 22,15
8360 FLASH 1
8370 PRINT "Please wait"
8380 FLASH 0
8399 END DEFine title
```

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.

Procedure *draw_scene*

Procedure *draw_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 not really 'speed critical' and our main concern in writing it should be to produce a clear procedure that is easy to modify.

```
2000 DEFine PROCedure draw_scene
2010 PAPER 2
2020 CLS
2040 FOR X=0 TO 31
2050  graph X,7,9
2060  graph X,12,9
2070  graph X,17,9
2080  graph X,22,9
2090  graph X,27,9
2100 END FOR X
```

```
2120 FOR Y=0 TO 2
2125  FOR X=0 TO 31
2130  graph X,Y,10
2135  END FOR X
2140 END FOR Y
2160 shafts 2,3,6
2170 shafts 1,8,11
2180 shafts 2,13,16
2190 shafts 1,18,21
2200 shafts 2,22,26
2220 graph XM%,YM%,11
2240 graph 1,26,5:graph 2,26,7
2250 graph 1,27,6:graph 2,27,8
2270 FOR I=0 TO 4
2280  graph A%(I,1),A%(I,2),A1%(I,1)
2285  graph A%(I,1)+1,A%(I,2),A1%(I,2)
2290 END FOR I
2300 SDATE 2000,1,1,0,0,0
2999 END DEFine draw_scene
```

Lines 2010 and 2020 clear the screen to red and then lines 2040 to 2100 print the horizontal tunnels in black. This creates five black bands through the red background. Next, lines 2120 to 2140 print a block of blue spaces to represent the sky. After this, all that remains is to print the vertical shafts that connect the tunnels. This is a problem that is best solved by calling another procedure – *shafts* (NO,Y1,Y2) – which will draw NO shafts vertically, each one starting at Y1 and ending at Y2. So, for example, *shafts*(2,7,9) draws two shafts between the tunnels at Y=6 and Y=10. Lines 2160 to 2200 use the yet-to-be-written procedure *shafts* to connect the tunnels.

Line 2220 prints 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 2240 and 2250 print the nest at the far left-hand end of the bottom tunnel. Finally, lines 2270 to 2290 print the ants and line 2300 zeros the time ready for the game to start.

The only procedure used by *draw_scene* is *shafts* (NO,Y1,Y2) and this has to be completed before moving on to the rest of the program:

```
8400 DEFine PROCedure shafts (NO,Y1,Y2)
8410 X1%=RND(0 TO 8)+6:X2%=RND(0 TO 10)+15
8420 FOR Y=Y1 TO Y2
8430  IF NO=1 THEN graph INT((X1%+X2%)/2),Y,9
8440  IF NO=2 THEN graph X1%,Y,9:graph X2%,Y,9
8450 END FOR Y
8499 END DEFine shafts
```

Line 8410 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 8440 within the FOR loop (lines 8420 to 8450). If only one shaft is required then it is printed at the average of X1% and X2%, which is a random position tending roughly towards the middle of the screen (line 8430).

Procedure *one_move* - the heart of the game

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

```
4000 DEFine PROCedure one_move
4010 move_man
4020 IF HIT=FALSE THEN
4030  move_ant
4040  get_time
4050 END IF
4099 END DEFine one_move
```

The job of procedure *move_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, procedure *move_ant* is concerned with moving the ants and checking to see if the man has been captured. Procedure *get_time* retrieves the time since the start of the game and checks to see if the allotted time is up.

Clearly, the way that procedure *one_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. Procedure *move_man* is the next procedure to be written:

```
5000 DEFine PROCedure move_man
5001 LOCal I%
5003 I%=KEYROW(1)
5004 IF I%=0 THEN END DEFine move_man
5010 IF YM%>4 THEN
5015  graph XM%,YM%,9
```

```
5020 ELSE
5022  graph XM%,YM%,10
5025 END IF
5060 IF I%=2 AND XM%>1 THEN XM%=XM%-1
5070 IF I%=16 AND XM%<30 THEN XM%=XM%+1
5080 IF I%=128 AND pix(YM%+1,XM%,1,1)=0 THEN
     down_shaft
5100 IF YM%>4 THEN
5102  graph XM%,YM%,4
5104 ELSE
5106  graph XM%,YM%,11
5107 END IF
5110 IF XM%=3 AND YM%=27 THEN HOME=TRUE
5199 END DEFine move_man
```

Line 5010 checks to see what the vertical position of the man is. If he is still above ground then a blue blank is printed by line 5022, otherwise a black blank is printed by line 5015. Thus in either case the graphics character of the man is removed from the screen.

Lines 5060 to 5080 then check each of the three possible arrow keys in turn. The value stored in I% has already been set by the use of KEYROW in line 5003. Line 5060 checks for the left arrow key, line 5070 checks for the right arrow key and line 5080 checks for the down arrow key. The only other point worth mentioning is that lines 5060 and 5070 also make sure that the attempted move would not take the man off the edge of the screen. Line 5080 calls procedure *down_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 *pix* (see Chapter One) is used to discover the colour of the character location. Procedure *move_man* finishes by reprinting the man character at the new position given by XM%, YM% (lines 5100 to 5107) and then checks to see if the nest has been reached (line 5110).

The only procedure left to tackle for *move_man* to work is *down_shaft*:

```
5500 DEFine PROCedure down_shaft
5520 FOR I=1 TO 4
5530  YM%=YM%+1
5540  graph XM%,YM%,4
5550  BEEP .1,YM%
5560  graph XM%,YM%,9
5570 END FOR I
5575 YM%=YM%+1
5580 BEEP
5599 END DEFine down_shaft
```

Lines 5520 to 5570 move the man down the shaft one character location at a time. Each move is accompanied by a sound produced by line 5550.

Procedure *move_ant* tackles the first real problem that has to be solved. The trouble is that there are one man and five ants. If procedure *move_ant* moved 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 procedure *move_ant* does:

```
3000 DEFine PROCedure move_ant
3020 Q%=RND(1 TO 5)-1
3030 graph A%(Q%,1),A%(Q%,2),9:
     graph A%(Q%,1)+1,A%(Q%,2),9
3040 IF RND<.01 THEN reverse Q%
3050 A%(Q%,1)=A%(Q%,1)+A%(Q%,3)*RND(1 TO 3)
3060 IF A%(Q%,1)<5 THEN
3061  A%(Q%,1)=5
3062  reverse Q%
3065 END IF
3070 IF A%(Q%,1)>28 THEN
3071  A%(Q%,1)=28
3072  reverse Q%
3075 END IF
3080 graph A%(Q%,1),A%(Q%,2),A1%(Q%,1)
3085 graph A%(Q%,1)+1,A%(Q%,2),A1%(Q%,2)
3090 HIT=got_man(Q%)
3099 END DEFine move_ant
```

Line 3020 sets Q% randomly to a number between 0 and 4 to determine which ant will be moved. Then line 3030 erases the ant from its current position. Line 3040 calls procedure *reverse(Q%)* at random to reverse the direction of ant Q%. RND will be smaller than .01 on average once in 100 calls to procedure *move_ant*.

Line 3050 increases or decreases the ant's X co-ordinate depending on the value stored in A%(Q,3). If A%(Q,3) is +1 then RND(1 TO 3) is added to the X co-ordinate, so moving the ant to the right; if A%(Q,3) is −1 then RND(1 TO 3) is subtracted from the X co-ordinate, so moving the ant to the left. The amount the ant moves is given by the value of RND(1 TO 3). This means that the ant can move by 1, 2 or 3 character locations at a time. Although in general moving anything more than one character location at a time is a potential source of trouble, this is really the only way that the impression that the ants move quickly can be achieved. After the ant's X co-ordinate

is updated it is checked by lines 3060 to 3075 to see if the ant is about to go off either edge of the screen. If so, procedure *reverse(Q%)* is called to change the ant's direction. Finally, lines 3080 and 3085 reprint the ant at its new position and line 3090 uses *got_man(Q%)* to check whether or not the ant just moved has captured the man or not.

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

```
3500 DEFine PROCedure reverse (Q%)
3510 IF A1%(Q%,1)=3 THEN
3520  A1%(Q%,1)=0
3525  A1%(Q%,2)=1
3530 ELSE
3540  A1%(Q%,1)=3
3545  A1%(Q%,2)=2
3550 END IF
3560 A%(Q%,3)=-A%(Q%,3)
3599 END DEFine reverse
```

The function *FNgot_man* is also easy to write:

```
8500 DEFine FuNction got_man (Q%)
8510 IF A%(Q%,2)<>YM% THEN RETurn FALSE
8520 IF A%(Q%,1)<>XM% AND A%(Q%,1)+1<>XM%
     THEN RETurn FALSE
8530 RETurn TRUE
8599 END DEFine got_man
```

The only remaining procedure used by procedure *one_move* is procedure *get_time*:

```
4500 DEFine PROCedure get_time
4510 TIME$=DATE$
4520 SEC=TIME$(17)*60
4530 SEC=SEC+TIME$(19 TO 20)
4550 IF SEC>MAX THEN TIME_UP=TRUE
4999 END DEFine get_time
```

Lines 4510 to 4530 get the time in seconds into SEC and then line 4550 tests to see if the allotted time is up and the game is at an end.

Now that *one_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.

Procedure *end_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 procedure *end_game* has to identify each and call appropriate procedures:

```
6000 DEFine PROCedure end_game
6010 IF HIT THEN
6012  capture
6014 ELSE
6020  IF TIME_UP THEN
6022   hatch
6024  ELSE
6030   IF HOME THEN
6032    destroy_nest
6040   END IF
6050  END IF
6060 END IF
6100 REPeat an_game
6110  AT 24,3
6112  PRINT "Another game Y/N ?";
6114  INPUT N$
6120  IF N$(1)="Y" OR N$(1)="N" THEN
     EXIT an_game
6125 END REPeat an_game
6130 IF N$(1)="Y" THEN AGAIN=TRUE
6199 END DEFine end_game
```

Procedure *capture* is called if an ant has reached the same screen location as the man. Procedure *hatch* is called if the game ends due to the time limit being reached and procedure *destroy_nest* is called if the man reaches the nest within the time limit. Notice the way that 'nested' IF statements have been used to select which procedure should be called. This ensures that no matter what the values contained in HIT, TIME_UP, or HOME are, only one of the three procedures is called. Following a call to one of these procedures, lines 6100 to 6130 ask the player if they want another game and set AGAIN accordingly.

Of the three procedures called by *end_game* the most complicated is *capture* and so a description of this is left until last. Procedure *hatch* simply prints ants at random positions tending toward the centre of the screen accompanied by descending scale:

```
6500 DEFine PROCedure hatch
6510 AT 23,3 :PRINT "THE ANTS ARE HATCHING "
```

```
6520 FOR I=1 TO RND(0 TO 20)+20
6530  X=RND(0 TO 25)+5
6532  Y=RND(0 TO 15)+5
6535  T%=RND(0 TO 4)
6540  graph X,Y,A1%(T%,1)
6545  graph X+1,Y,A1%(T%,2)
6550  BEEP .1,I
6555  PAUSE 1
6560 END FOR I
6570 BEEP
6599 END DEFine hatch
```

Procedure *destroy nest* rewards the player with a simple 'fanfare':

```
6800 DEFine PROCedure destroy_nest
6810 AT 23,2: PRINT "YOU DID IT"
6815 RESTORE 6900
6820 REPeat tune
6830  READ D,P
6840  IF D=999 THEN EXIT tune
6850  BEEP D,P
6860  PAUSE D*125
6865  BEEP
6870 END REPeat tune
6900 DATA .1,14,.1,13
6910 DATA .1,12,.1,11
6920 DATA .2,10,.2,10
6990 DATA 999,999
6999 END DEFine destroy_nest
```

The notes of the fanfare are specified by the values in the DATA statements (lines 6900 to 6990). Each note corresponds to a pair of values, the first giving the duration and the second the pitch. A duration of 999 is used to mark the end of the sequence of notes and this is detected by line 6840.

The idea that procedure *capture* implements 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 complicated sequence of events but in practice it is not so difficult:

```
6200 DEFine PROCedure capture
6210 AT 23,3: PRINT "You are captured!"
6220 REPeat ant
6230  IF Q%=4 THEN EXIT ant
6240  goto_man(Q%)
6250  find_shafts A%(Q%,2)
```

```
6260  IF ABS(X2%-A%(Q%,1))<ABS(X1%-A%(Q%,1))
      AND X2%<>0 THEN
6262   drag X2%
6264  ELSE
6266   drag X1%
6268  END IF
6270  graph A%(Q%,1)+2,A%(Q%,2),9
6275  graph A%(Q%,1)+3,A%(Q%,2),9
6280  BEEP .1,-10
6290  down_shaft
6300  graph XM%,YM%,4
6310  Q%=Q%+1
6320 END REPeat ant
6330 goto_man(Q%)
6340 drag 5
6399 END DEFine capture
```

Procedure *goto_man(Q%)* (line 6240) moves ant Q% to the current position of the man. Once the ant has reached the man, line 6250 calls procedure *find_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 6260 tests to discover which is closer to the current position of the ant that has the man. Then procedure *drag* is called to move the ant plus the man to the closest shaft. Finally procedure *down_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 nest (line 6340).

Of course, the simplicity of procedure *capture* is due to its use of a number of new procedures. Procedure *goto_man(Q%)* simply 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 DEFine PROCedure goto_man (Q%)
5910 IF A%(Q%,1)+1=XM% THEN END DEFine goto_man
5920 IF SGN(XM%-A%(Q%,1)-1)<>A%(Q%,3) THEN
     reverse Q%
5930 REPeat mve_ant
5940  graph A%(Q%,1),A%(Q%,2),9
5945  graph A%(Q%,1)+1,A%(Q%,2),9
5950  A%(Q%,1)=A%(Q%,1)+A%(Q%,3)
5960  graph A%(Q%,1),A%(Q%,2),A1%(Q%,1)
5970  graph A%(Q%,1)+1,A%(Q%,2),A1%(Q%,2)
5975  delay
5980  IF A%(Q%,1)+1=XM% THEN EXIT mve_ant
5990 END REPeat mve_ant
5999 END DEFine goto_man
```

Line 5920 checks to see if the ant is already facing toward the man; if not, procedure *reverse(Q%)* is called. After this, the REPEAT loop (lines 5930 to 5980) moves the ant one location at a time until it is in contact with the man. Line 5920 uses the SGN(X) function which returns −1 if X is negative, +1 if it is positive and 0 if it is zero. This function is standard on most versions of BASIC but it appears to be missing from SuperBASIC. However, this is not very much of a problem as it is easy to add:

```
8700 DEFine FuNction SGN(N%)
8710 IF N%=0 THEN
8715  RETurn 0
8716 ELSE
8720  IF N%<0 THEN
8725   RETurn -1
8730  ELSE
8740   RETurn 1
8750  END IF
8760 END IF
8799 END DEFine SGN
```

Procedure *find_shafts* works by examining the row of character locations just below the ant's tunnel and storing the co-ordinates of any that are back in X1% and X2%.

```
5600 DEFine PROCedure find_shafts (Y%)
5610 X%=0
5620 REPeat find1
5630  COL=pix(Y%+1,X%,1,1)
5640  X1%=X%
5650  X%=X%+1
5660  IF X%>28 OR COL=0 THEN EXIT find1
5670 END REPeat find1
5680 X2%=0
5690 REPeat find2
5700  COL=pix(Y%+1,X%,1,1)
5705  IF COL=0 THEN X2%=X%
5710  X%=X%+1
5720  IF X%>28 OR COL=0 THEN EXIT find2
5730 END REPeat find2
5799 END DEFine find_shafts
```

The first REPEAT loop (lines 5620 to 5670) searches for the first shaft and the second REPEAT loop (line 5690 to 5730) 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 *capture* is procedure *drag*, which is very similar to procedure *goto_man*.

```
5800 DEFine PROCedure drag (X%)
5810 IF A%(Q%,1)+2=X% THEN XM%=X%:
     END DEFine drag
5820 IF SGN(X%-A%(Q%,1)-2)<>A%(Q%,3) THEN
     reverse Q%
5830 REPeat goto_shaft
5840  graph A%(Q%,1),A%(Q%,2),9
5845  graph A%(Q%,1)+1,A%(Q%,2),9
5848  graph A%(Q%,1)+2,A%(Q%,2),9
5850  A%(Q%,1)=A%(Q%,1)+A%(Q%,3)
5860  graph A%(Q%,1),A%(Q%,2),A1%(Q%,1)
5863  graph A%(Q%,1)+1,A%(Q%,2),A1%(Q%,2)
5866  graph A%(Q%,1)+2,A%(Q%,2),4
5870  delay
5880  IF A%(Q%,1)+2=X% THEN EXIT goto_shaft
5890 END REPeat goto_shaft
5895 XM%=X%
5899 END DEFine drag
```

The only real difference between *drag* and procedure *goto_man* is that procedure *drag* moves the ant and the man to the nearest shaft whereas procedure *goto_man* moves only the ant to the position of the man. To allow the player enough time to see what is happening, *goto_man* and *drag* both use procedure *delay* to slow things down a little:

```
8600 DEFine PROCedure delay
8610 PAUSE 5
8699 END DEFine delay
```

Making the game better

After playing a few games of Ant Hill it quickly becomes obvious that 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 the idea of 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 try 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 decreasing the time allowed has very little effect until you suddenly

reach the point where the game is almost impossible. The trouble is that the game can always be completed in about 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 have first to 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 toward the right and as long as the man makes a dash for the left-hand tunnels, he can get to the nest 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 procedure *reverse* this is most easily achieved by adding the following line:

```
1135  IF RND>.5 THEN reverse 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 if the man moves onto a position that is already occupied by an ant. This can be done by adding:

```
5090 IF pix(YM%,XM%,4,4)=3 THEN
5095  HIT=TRUE
5096  Q%=(YM%-2)/5-1
5098 END IF
```

to procedure *move_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 3050 in procedure *move_ant* to:

```
3050 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 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 the 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 3060, 3061, 3070 and 3071 in procedure *move_ant* have to be changed to:

```
3060 IF A%(Q%,1)<10 THEN
3061   A%(Q%,1)=10
3070 IF A%(Q%,1)>20 THEN
3071   A%(Q%,1)=20
```

and line 8410 in procedure *shafts* changed to:

```
8410 X1%=RND(1 TO 4)+9:X2%=RND(0 TO 4)+16
```

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

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. Procedure *destroy_nest* and procedure *hatch* are both weak when compared to the fascinating performance that the ants go through when procedure capture 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 the modifications introduced in the text.

```
10 REMark ANT HILL
20 MODE 8
25 WINDOW 512,256,0,0
26 title
30 init_graph
```

```
 35 REPeat game
 36  init
 50  draw_scene
 60  REPeat main
 70   one_move
 80   IF TIME_UP OR HIT OR HOME THEN EXIT main
 90  END REPeat main
100  end_game
110  IF AGAIN=FALSE THEN EXIT game
120 END REPeat game
130 INK 0
140 PAPER 7
150 CLS
199 STOP

350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430 READ B
440 POKE_W I,B
450 END FOR I
520 C_TAB=RESPR(32*12)
530 def_graph 0,3,0,0,14,15,63,31,51,68,132
540 def_graph 1,3,0,3,20,216,252,168,192,64,16
550 def_graph 2,3,0,0,112,248,252,248,204,34,33
560 def_graph 3,3,0,192,40,27,63,25,3,4,8
570 def_graph 4,6,0,24,24,126,24,60,102,195,195
580 def_graph 5,4,0,1,1,3,3,7,7,15,15
590 def_graph 6,4,0,31,31,63,63,127,127,255,255
600 def_graph 7,4,0,128,128,192,192,224,224,240,240
610 def_graph 8,4,0,248,248,252,252,254,254,255,255
620 def_graph 9,0,0,255,255,255,255,255,255,255,255
630 def_graph 10,1,1,255,255,255,255,255,255,255,25
640 def_graph 11,6,1,24,24,126,24,60,102,195,195
650 FALSE=0
660 TRUE=1
670 AGAIN=TRUE
699 END DEFine init_graph

1000 DEFine PROCedure init
1050 RANDOMISE
1060 XM%=RND(0 TO 20)+10
1070 YM%=2
1080 DIM A%(4,3),A1%(4,2)
1090 FOR I=0 TO 4
```

```
1100  A%(I,1)=RND(0 TO 10)+10
1110  A%(I,2)=(I+1)*5+2
1120  A%(I,3)=1
1130  A1%(I,1)=0:A1%(I,2)=1
1135  IF RND>.5 THEN reverse I
1140 END FOR I
1160 AGAIN=FALSE
1170 HIT=FALSE
1180 HOME=FALSE
1190 TIME_UP=FALSE
1299 END DEFine init

1300 DEFine PROCedure def_graph(S%,F%,B%,
     R0%,R1%,R2%,R3%,R4%,R5%,R6%,R7%)
1310 LOCal A,FH%,FL%,BH%,BL%
1320 A=C_TAB+32*S%
1330 FH%=(F% DIV 4)*2
1340 FL%=F% && 3
1350 BH%=(B% DIV 4)*2
1360 BL%=B% && 3
1370 charline(R0%)
1380 charline(R1%)
1390 charline(R2%)
1400 charline(R3%)
1410 charline(R4%)
1420 charline(R5%)
1430 charline(R6%)
1440 charline(R7%)
1499 END DEFine def_graph

1500 DEFine PROCedure charline(R%)
1502 LOCal M%,J,I,DL%,DH%
1503 M%=128
1504 FOR J=1 TO 2
1505  DL%=0:DH%=0
1510  FOR I=1 TO 4
1520   IF (R% && M%)=M% THEN
1530    DL%=FH%+DL%*4
1540    DH%=FL%+DH%*4
1550   ELSE
1560    DL%=BH%+DL%*4
1570    DH%=BL%+DH%*4
1580   END IF
1590   M%=M% DIV 2
1600  END FOR I
1610  POKE A,DL%
1620  POKE A+1,DH%
1630  A=A+2
1640 END FOR J
1699 END DEFine charline
```

```
1700 DEFine PROCedure graph(X%,Y%,S%)
1710 CALL do_graph,X%,Y%,S%,C_TAB
1799 END DEFine graph

2000 DEFine PROCedure draw_scene
2010 PAPER 2
2020 CLS
2040 FOR X=0 TO 31
2050  graph X,7,9
2060  graph X,12,9
2070  graph X,17,9
2080  graph X,22,9
2090  graph X,27,9
2100 END FOR X
2120 FOR Y=0 TO 2
2125  FOR X=0 TO 31
2130  graph X,Y,10
2135  END FOR X
2140 END FOR Y
2160 shafts 2,3,6
2170 shafts 1,8,11
2180 shafts 2,13,16
2190 shafts 1,18,21
2200 shafts 2,22,26
2220 graph XM%,YM%,11
2240 graph 1,26,5:graph 2,26,7
2250 graph 1,27,6:graph 2,27,8
2270 FOR I=0 TO 4
2280  graph A%(I,1),A%(I,2),A1%(I,1)
2285  graph A%(I,1)+1,A%(I,2),A1%(I,2)
2290 END FOR I
2300 SDATE 2000,1,1,0,0,0
2999 END DEFine draw_scene

3000 DEFine PROCedure move_ant
3020 Q%=RND(1 TO 5)-1
3030 graph A%(Q%,1),A%(Q%,2),9:
     graph A%(Q%,1)+1,A%(Q%,2),9
3040 IF RND<5E-2 THEN reverse Q%
3050 A%(Q%,1)=A%(Q%,1)+A%(Q%,3)*2
3060 IF A%(Q%,1)<10 THEN
3061  A%(Q%,1)=10
3062  reverse Q%
3065 END IF
3070 IF A%(Q%,1)>20 THEN
3071  A%(Q%,1)=20
3072  reverse Q%
3075 END IF
3080 graph A%(Q%,1),A%(Q%,2),A1%(Q%,1)
3085 graph A%(Q%,1)+1,A%(Q%,2),A1%(Q%,2)
```

```
3090 HIT=got_man(Q%)
3099 END DEFine move_ant

3500 DEFine PROCedure reverse (Q%)
3510 IF A1%(Q%,1)=3 THEN
3520  A1%(Q%,1)=0
3525  A1%(Q%,2)=1
3530 ELSE
3540  A1%(Q%,1)=3
3545  A1%(Q%,2)=2
3550 END IF
3560 A%(Q%,3)=-A%(Q%,3)
3599 END DEFine reverse

4000 DEFine PROCedure one_move
4010 move_man
4020 IF HIT=FALSE THEN
4030  move_ant
4040  get_time
4050 END IF
4099 END DEFine one_move

4500 DEFine PROCedure get_time
4510 TIME$=DATE$
4520 SEC=TIME$(17)*60
4530 SEC=SEC+TIME$(19 TO 20)
4550 IF SEC>MAX THEN TIME_UP=TRUE
4999 END DEFine get_time

5000 DEFine PROCedure move_man
5001 LOCal I%
5003 I%=KEYROW(1)
5004 IF I%=0 THEN END DEFine move_man
5010 IF YM%>4 THEN
5015  graph XM%,YM%,9
5020 ELSE
5022  graph XM%,YM%,10
5025 END IF
5060 IF I%=2 AND XM%>1 THEN XM%=XM%-1
5070 IF I%=16 AND XM%<30 THEN XM%=XM%+1
5080 IF I%=128 AND pix(YM%+1,XM%,1,1)=0 THEN
     down_shaft
5090 IF pix(YM%,XM%,4,4)=3 THEN
5095  HIT=TRUE
5096  Q%=(YM%-2)/5-1
5098 END IF
5100 IF YM%>4 THEN
5102  graph XM%,YM%,4
5104 ELSE
5106  graph XM%,YM%,11
```

```
5107 END IF
5110 IF XM%=3 AND YM%=27 THEN HOME=TRUE
5199 END DEFine move_man

5500 DEFine PROCedure down_shaft
5520 FOR I=1 TO 4
5530  YM%=YM%+1
5540  graph XM%,YM%,4
5550  BEEP .1,YM%
5560  graph XM%,YM%,9
5570 END FOR I
5575 YM%=YM%+1
5580 BEEP
5599 END DEFine down_shaft

5600 DEFine PROCedure find_shafts (Y%)
5610 X%=0
5620 REPeat find1
5630  COL=pix(Y%+1,X%,1,1)
5640  X1%=X%
5650  X%=X%+1
5660  IF X%>28 OR COL=0 THEN EXIT find1
5670 END REPeat find1
5680 X2%=0
5690 REPeat find2
5700  COL=pix(Y%+1,X%,1,1)
5705  IF COL=0 THEN X2%=X%
5710  X%=X%+1
5720  IF X%>28 OR COL=0 THEN EXIT find2
5730 END REPeat find2
5799 END DEFine find_shafts

5800 DEFine PROCedure drag (X%)
5810 IF A%(Q%,1)+2=X% THEN XM%=X%:
     END DEFine drag
5820 IF SGN(X%-A%(Q%,1)-2)<>A%(Q%,3) THEN
     reverse Q%
5830 REPeat goto_shaft
5840  graph A%(Q%,1),A%(Q%,2),9
5845  graph A%(Q%,1)+1,A%(Q%,2),9
5848  graph A%(Q%,1)+2,A%(Q%,2),9
5850  A%(Q%,1)=A%(Q%,1)+A%(Q%,3)
5860  graph A%(Q%,1),A%(Q%,2),A1%(Q%,1)
5863  graph A%(Q%,1)+1,A%(Q%,2),A1%(Q%,2)
5866  graph A%(Q%,1)+2,A%(Q%,2),4
5870  delay
5880  IF A%(Q%,1)+2=X% THEN EXIT goto_shaft
5890 END REPeat goto_shaft
5895 XM%=X%
5899 END DEFine drag
```

```
5900 DEFine PROCedure goto_man (Q%)
5910 IF A%(Q%,1)+1=XM% THEN END DEFine goto_man
5920 IF SGN(XM%-A%(Q%,1)-1)<>A%(Q%,3) THEN
     reverse Q%
5930 REPeat mve_ant
5940  graph A%(Q%,1),A%(Q%,2),9
5945  graph A%(Q%,1)+1,A%(Q%,2),9
5950  A%(Q%,1)=A%(Q%,1)+A%(Q%,3)
5960  graph A%(Q%,1),A%(Q%,2),A1%(Q%,1)
5970  graph A%(Q%,1)+1,A%(Q%,2),A1%(Q%,2)
5975  delay
5980  IF A%(Q%,1)+1=XM% THEN EXIT mve_ant
5990 END REPeat mve_ant
5999 END DEFine goto_man

6000 DEFine PROCedure end_game
6010 IF HIT THEN
6012  capture
6014 ELSE
6020  IF TIME_UP THEN
6022   hatch
6024  ELSE
6030   IF HOME THEN
6032    destroy_nest
6040   END IF
6050  END IF
6060 END IF
6100 REPeat an_game
6110  AT 24,3
6112  PRINT "Another game Y/N ?";
6114  INPUT N$
6120  IF N$(1)="Y" OR N$(1)="N" THEN
     EXIT an_game
6125 END REPeat an_game
6130 IF N$(1)="Y" THEN AGAIN=TRUE
6199 END DEFine end_game

6200 DEFine PROCedure capture
6210 AT 23,3: PRINT "You are captured!"
6220 REPeat ant
6230  IF Q%=4 THEN EXIT ant
6240  goto_man(Q%)
6250  find_shafts A%(Q%,2)
6260  IF ABS(X2%-A%(Q%,1))<ABS(X1%-A%(Q%,1))
      AND X2%<>0 THEN
6262   drag X2%
6264  ELSE
6266   drag X1%
6268  END IF
6270  graph A%(Q%,1)+2,A%(Q%,2),9
```

```
6275  graph A%(Q%,1)+3,A%(Q%,2),9
6280  BEEP .1,-10
6290  down_shaft
6300  graph XM%,YM%,4
6310  Q%=Q%+1
6320 END REPeat ant
6330 goto_man(Q%)
6340 drag 5
6399 END DEFine capture

6500 DEFine PROCedure hatch
6510 AT 23,3 :PRINT "THE ANTS ARE HATCHING "
6520 FOR I=1 TO RND(0 TO 20)+20
6530  X=RND(0 TO 25)+5
6532  Y=RND(0 TO 15)+5
6535  T%=RND(0 TO 4)
6540  graph X,Y,A1%(T%,1)
6545  graph X+1,Y,A1%(T%,2)
6550  BEEP .1,I
6555  PAUSE 1
6560 END FOR I
6570 BEEP
6599 END DEFine hatch

6800 DEFine PROCedure destroy_nest
6810 AT 23,2: PRINT "YOU DID IT"
6815 RESTORE 6900
6820 REPeat tune
6830  READ D,P
6840  IF D=999 THEN EXIT tune
6850  BEEP D,P
6860  PAUSE D*125
6865  BEEP
6870 END REPeat tune
6900 DATA .1,14,.1,13
6910 DATA .1,12,.1,11
6920 DATA .2,10,.2,10
6990 DATA 999,999
6999 END DEFine destroy_nest

8000 DEFine PROCedure title
8010 BORDER 0:INK 6:PAPER 0:CLS
8020 AT 3,14 :PRINT "A N T  H I L L"
8040 AT 7,3 : PRINT "In this game you have a
     fixed "
8050 PRINT TO 2;"amount of time to destroy
     the ants"
8060 PRINT TO 2;"nest before the ants hatch!"\
8070 PRINT TO 2;"BEWARE: the soldier ants
     guarding the"
```

```
8080 PRINT TO 2;"tunnels will capture you and
     take you"
8090 PRINT TO 2;"to the nest to feed the young!"\

8100 PRINT TO 2;"Good Luck!"
8200 REPeat diff
8300  AT 18,5 :PRINT "Which difficulty level -"
8310  PRINT TO 3;"1. Expert, 2. Medium or
      3. Novice ";
8320  INPUT DF
8330  IF DF>0 AND DF<4 THEN EXIT diff
8335 END REPeat diff
8340 MAX=(20+DF*10)
8350 AT 22,15
8360 FLASH 1
8370 PRINT "Please wait"
8380 FLASH 0
8399 END DEFine title

8400 DEFine PROCedure shafts (NO,Y1,Y2)
8410 X1%=RND(1 TO 4)+9:X2%=RND(0 TO 4)+16
8420 FOR Y=Y1 TO Y2
8430  IF NO=1 THEN graph INT((X1%+X2%)/2),Y,9
8440  IF NO=2 THEN graph X1%,Y,9:graph X2%,Y,9
8450 END FOR Y
8499 END DEFine shafts

8500 DEFine FuNction got_man (Q%)
8510 IF A%(Q%,2)<>YM% THEN RETurn FALSE
8520 IF A%(Q%,1)<>XM% AND A%(Q%,1)+1<>XM%
     THEN RETurn FALSE
8530 RETurn TRUE
8599 END DEFine got_man

8600 DEFine PROCedure delay
8610 PAUSE 5
8699 END DEFine delay

8700 DEFine FuNction SGN(N%)
8710 IF N%=0 THEN
8715  RETurn 0
8716 ELSE
8720  IF N%<0 THEN
8725   RETurn -1
8730  ELSE
8740   RETurn 1
8750  END IF
8760 END IF
8799 END DEFine SGN
```

```
9500 DEFine FuNction pix(R%,C%,X%,Y%)
9510 LOCal A%,P%,B%,A
9520 P%=C%*8+X%
9530 A=131072+(512*R%+Y%*64+P% DIV 4)*2
9540 B%=PEEK(A+1)
9550 A%=PEEK(A)
9560 P%=P% MOD 4
9570 IF P%=3 THEN  RETurn ((2 && A%)*2)+(3 && B%)
9580 IF P%=2 THEN  RETurn ((8 && A%)DIV 2)+
     ((12 && B%) DIV 4)
9590 IF P%=1 THEN  RETurn ((32 && A%)DIV 8)+
     ((48 && B%) DIV 16)
9600 IF P%=0 THEN  RETurn ((128 && A%)DIV 32)+
     ((192 && B%) DIV 64)
9699 END DEFine pix
```

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 (i.e. diagonally) 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 Break Out and Leap Frog is to be found in the path that the frog takes as it flies through the air. In Break Out 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 gravity. Each time the frog lands on the seesaw the frog that 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.

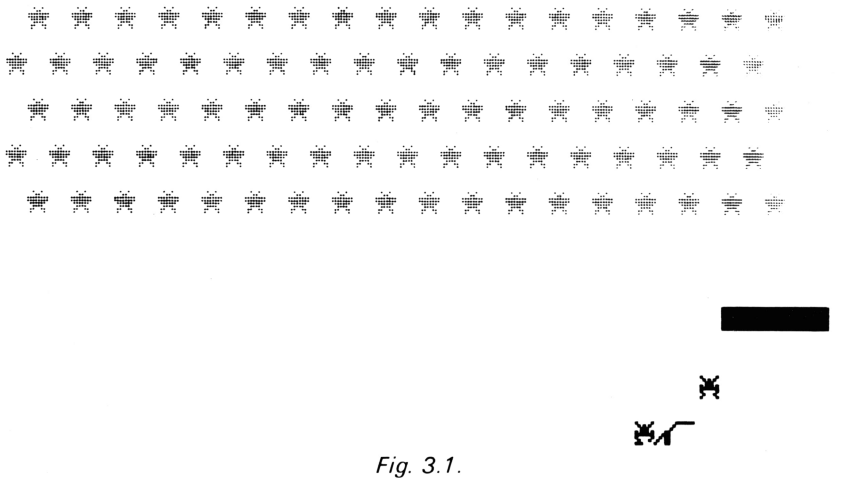


Fig. 3.1.

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:

```
graph X,Y,B          :REM erase object
X=X+VX:Y=Y+VY        :REM update co-ordinates
graph X,Y,S          :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 screen. 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 init_graph
 15 PAPER 0:CLS
 20 VX=1:VY=1
 30 X=RND(0 TO 31):Y=RND(0 TO 31)
 40 graph X,Y,0
 50 X=X+VX:Y=Y+VY
 60 IF X<1 OR X>30 THEN VX=-VX
 70 IF Y<1 OR Y>30 THEN VY=-VY
 80 graph X,Y,1
 90 GO TO 40

350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430  READ B
440  POKE_W I,B
450 END FOR I
510 C_TAB=RESPR(32*2)
520 def_graph 0,0,0,255,255,255,255,255,255,255,255,255
530 def_graph 1,7,7,255,255,255,255,255,255,255,255,255
599 END DEFine init_graph
```

(Remember to include procedure *graph*!) Line 60 detects collisions with the vertical edges of the screen and reverses the horizontal velocity. Line 70 detects collisions with the horizontal edge of the screen and reverses the vertical velocity. The overall result is a solid block 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 init_graph
 15 PAPER 0:CLS
 20 X=0:Y=31
 30 VX=.5:VY=-.8
 40 A=.025
 50 graph X,Y,0
 60 X=X+VX:Y=Y+VY
 70 VY=VY+A
 80 IF Y>30 THEN STOP
 90 graph X,Y,1
100 GO TO 50
```

(Remember to include procedure *init_graph*, as used in the previous program, and *graph*.) 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 solid block that follows a parabolic path. The size of the constant used in line 40 controls the force of 'gravity' that the solid block responds to. Increasing it will bring the block 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. The elements of the game are straightforward. The player is given ten frogs to attempt to eat all the insects printed at the top of the screen. Each new frog starts off by jumping off 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' the 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 REMark Leap Frog
 20 WINDOW 512,256,0,0
 25 title
 30 init_graph
 35 REPeat game
```

```
 36  init
 50  print_flies
 60  startup
 70  REPeat main
 80   move_ss
 90   move_f
100   IF GAME_END THEN EXIT main
110  END REPeat main
120  end_game
130  IF NOT AGAIN THEN EXIT game
140 END REPeat game
199 STOP
```

Procedure *title* prints a screen of instructions and asks the user to choose a difficulty level. Procedure *init_graph* sets up the machine code needed for *graph* and the user-defined graphics. Procedure *init* initialises variables used later in the game. Procedure *print_flies* prints five rows of insects at the top of the screen ready for the game to commence. Procedure *startup* 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 70 to 110. Procedure *move_ss* allows the player to move the seesaw using the left and right arrow keys and procedure *move_f* moves the frog, taking account of gravity and any bouncing that is necessary. Finally, procedure *end_game* sums up the player's performance and asks if another game is required.

Procedures *init_graph, init* and *title*

Procedure *init_graph* is not very different from that given in Chapter Two, only now it defines the graphics characters needed for Leap Frog.

```
350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430  READ B
440  POKE_W I,B
450 END FOR I
510 C_TAB=RESPR(32*9)
520 def_graph 0,2,6,130,84,56,186,254,130,68,255
```

```
530 def_graph 1,2,6,1,2,4,24,56,88,152,24
540 def_graph 2,2,6,252,0,0,0,0,0,0,0
550 def_graph 3,1,6,36,24,255,255,126,60,102,66
560 def_graph 4,4,6,130,84,56,186,254,130,68,130
570 def_graph 5,6,6,255,255,255,255,255,255,255,255
575 def_graph 6,2,2,255,255,255,255,255,255,255,255
576 def_graph 7,2,6,128,64,32,24,28,26,25,24
577 def_graph 8,2,6,63,0,0,0,0,0,0,0
580 FALSE=0
590 TRUE=1
599 END DEFine init_graph
```

Lines 520 to 577 define the graphics characters used by Leap Frog, which are illustrated in Fig. 3.2 and summarised in Table 3.1.

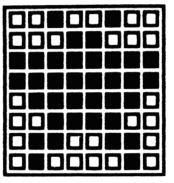
Table 3.1 Graphics characters

Number	Shape	Foreground	Background
0	frog and lower seesaw	red	yellow
1	right seesaw swivel	red	yellow
2	right upper seesaw	red	yellow
3	fly	blue	yellow
4	frog	green	yellow
5	space	yellow	yellow
6	block	red	red
7	left seesaw swivel	red	yellow
8	left upper seesaw	red	yellow

Procedure *init* is very simple:

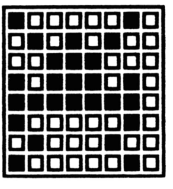
```
1000 DEFine PROCedure init
1010 M=1
1030 X%=23
1040 FROG=1
1050 GAME_END=FALSE
1060 AGAIN=FALSE
1070 SC=0
1099 END DEFine init
```

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 on the screen and if M is 0 the right seesaw is on the screen. X% gives the horizontal position of the seesaw and it is set to 23 by line 1030 to give the initial position of the seesaw when the game starts. GAME_END if used to signal the end of the game and SC is used to hold the current score.



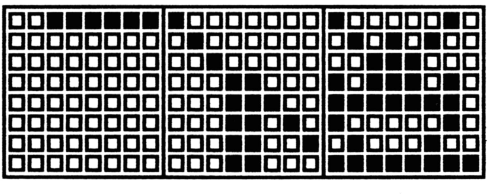
3

(a)



4

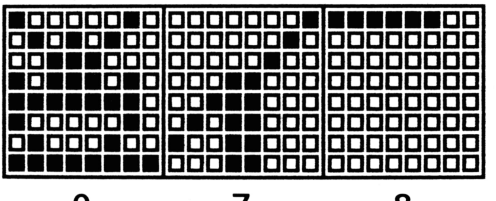
(b)



2 1 0

Right Seesaw

(c)



0 7 8

Left Seesaw

Fig. 3.2. Graphics characters for (a) Insect and frog. (b) Right seesaw. (c) Left seesaw.

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

```
6000 DEFine PROCedure title
6010 PAPER 6
6015 INK 1
6020 CLS
6030 AT 5,10
6040 PRINT "L E A P  F R O G"
6050 AT 10,4
6060 PRINT "In this game you must eat the flies"\\
6070 PRINT TO 3;"by jumping your two frogs off the"\\

6080 PRINT TO 3; "seesaw using the left and right
     arrow"\\
6090 PRINT TO 3; "keys - you have ten frogs to
     clear"\\
6100 PRINT TO 3;"the screen ....."
6110 REPeat diff
6120  AT 21,5
6130  PRINT "Which difficulty level "
6140  PRINT TO 8;"1 (difficult) to 10 (easy)";
6150  INPUT D
6160  IF D>0 AND D<11 THEN EXIT diff
6170 END REPeat diff
6199 END DEFine title
```

Procedure *print_flies*

Procedure *print_flies* is a simple procedure that prints five rows of insects:

```
2000 DEFine PROCedure print_flies
2010 CLS
2020 C%=0
2030 FOR V=2 TO 10 STEP 2
2040  C%=NOT C%
2050  FOR U=2 TO 30 STEP 2
2060   graph U-C%,V,3
2070  END FOR U
2080 END FOR V
2099 END DEFine print_flies
```

The only interesting technique is the use of C% in lines 2040 to 2080 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.

Procedure *startup*

After *print_flies startup* 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 *startup* is called again from within the animation loop.

```
5000 DEFine PROCedure startup
5010 move_ss
5020 FOR I=26 TO 31
5030  graph I,15,6
5040 END FOR I
5050 FOR F=31 TO 24 STEP -1
5060  graph F+1,14,5
5070  graph F,14,4
5080  move_ss
5090 END FOR F
5100 FOR G=14 TO 19
5110  graph 24,G-1,5
5120  graph 24,G,4
5130  move_ss
5140 END FOR G
5145 graph 24,19,5
5150 F%=24
5155 F=F%
5160 G%=19
5165 G=G%
5170 A=.5+RND/10
5180 B=-(.2+RND/2.5)
5190 P=9E-3
5200 AT 21,2
5210 PRINT "FROG ";FROG;" SCORE=";SC
5220 A=A+A/D
5230 B=B+B/D
5240 seesaw
5299 END DEFine start_up
```

Line 5010 calls *move_ss*, which in this case simply serves to draw the seesaw at its initial position. Then the first FOR loop, lines 5020 to 5040, 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 second FOR loop, lines 5050 to 5090, moves the frog along the ledge. Each time through the loop, *move_ss* is called, line 5080, to allow the player an opportunity to move the seesaw. The third FOR loop, lines 5100 to 5140, moves the frog vertically downward. Finally, lines 5150 to 5230 initialise variables used by the rest of the program. F and G are

used to record the current position of the frog. That is, the frog is at graph F,G,4. The variables A and B are the vertical and horizontal velocity of the frog. Both are set to random values – by lines 5170 and 5180 respectively. Line 5190 sets P, the 'gravity constant', to 0.009 (P=9E−3). Lines 5220 and 5230 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, *startup* calls *seesaw* to test if the falling frog has or has not landed on the seesaw.

Procedures *move_ss, print_ss* and *move_f*

Procedure *move_ss* allows the player to move the seesaw using the arrow keys.

```
3000 DEFine PROCedure move_ss
3005 LOCal A
3010 A=KEYROW(1)
3020 IF A=16 AND X%<28 THEN X%=X%+1
3030 IF A=2 AND X%>0 THEN X%=X%-1
3040 print_ss
3099 END DEFine move_ss
```

Lines 3020 and 3030 test for the right and left arrow keys and update the horizontal position of the seesaw (stored in X%). Line 3040 prints the seesaw at its new position. The actual printing of the seesaw is done by *print_ss*:

```
3100 DEFine PROCedure print_ss
3110 SELect ON M
3120  ON M=1
3130   graph X%,21,5
3140   graph X%+1,21,0
3150   graph X%+2,21,1
3180   graph X%+3,21,2
3220   graph X%+4,21,5
3240  ON M=0
3250   graph X%,21,5
3260   graph X%+1,21,8
3270   graph X%+2,21,7
3280   graph X%+3,21,0
3290   graph X%+4,21,5
3360 END SELect
3999 END DEFine print_ss
```

Either a right or a left seesaw is printed depending on the value of M.

Note that there is no need to blank out the entire seesaw when changing from left to right and vice versa. As it only moves one position to the left or right each time through the animation loop it is enough to blank both ends of it.

Procedure *move_f* is fairly straightforward and follows the theory for bouncing and falling objects given at the beginning of this chapter.

```
4000 DEFine PROCedure move_f
4010 graph F%,G%,5
4020 F=F+B
4030 G=G+A
4040 A=A+P
4050 IF F>30 OR F<2 THEN B=-B
4060 IF G<2 THEN G=2:A=-A
4070 G%=G
4080 F%=F
4090 IF G%<11 THEN hit_fly
4100 graph F%,G%,4
4110 IF G%>20 THEN seesaw
4199 END DEFine move_f
```

Line 4010 blanks out the frog at the old position. Lines 4020 to 4040 perform 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 4060 deals with bounces off the top edge of the screen. Line 4090 checks to see if the frog is high enough up the screen to have hit a fly and calls *hit_fly* if it is. The reprinting of the frog is taken care of by line 4100. Line 4110 checks to see if the frog is low enough on the screen to have landed on the seesaw and calls *seesaw* if this is so.

Procedures *seesaw, hit_fly* and *miss*

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

```
4500 DEFine PROCedure seesaw
4510 D%=F-X%-M
4520 IF D%<1 OR D%>2 THEN miss:END DEFine seesaw
4530 M=NOT M
4540 A=-A
4550 G=20
4560 F=M*3+X%
```

```
4570 G%=G
4575 F%=F
4580 BEEP .1,10
4590 PAUSE 5
4600 BEEP
4699 END DEFine seesaw
```

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 a 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 either 1 or 2 for a correct landing. This 'corrected' difference is calculated by line 4510 and the test for a correct landing is carried out by line 4520 which calls *miss* 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. Lines 4550 to 4575 update 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, *miss* is called to bring a new frog into play (unless, of course, all ten frogs have already been used).

```
8000 DEFine PROCedure miss
8010 graph F%,G%,5
8020 FROG=FROG+1
8030 IF FROG<10 THEN
8040  BEEP .1,-10
8050  PAUSE 6
8060  BEEP
8070  M=1
8080  startup
8090 ELSE
8100  GAME_END=TRUE
8110 END IF
8199 END DEFine miss
```

If ten frogs have been used, line 8100 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 8070 and then line 8080 calls *startup* to continue the game with a new frog.

Procedure *hit_fly* simply uses the *pix* function (described in Chapter One) to test the colour of a point within the character location that the frog is about to move onto. If it isn't yellow then the frog has hit a fly.

```
7000 DEFine PROCedure hit_fly
7010 IF pix(G%,F%,4,4)<>1 THEN END DEFine hit_fly
7020 BEEP .1,20
7030 PAUSE 2
7040 BEEP
7050 SC=SC+1
7060 AT 21,2
7070 PRINT "FROG ";FROG;" SCORE=";SC
7080 IF SC=76 THEN GAME_END=TRUE
7099 END DEFine hit_fly
```

If the frog has hit a fly then all that happens is that the score is updated (line 7050) and checked to see if all the flies have been eaten (line 7080).

Procedure *end_game*

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

```
9000 DEFine PROCedure end_game
9010 CLS
9020 AT 10,3
9030 SELect ON SC
9040  ON SC=0 TO 10
9050   PRINT "A mere tadpole could do better"
9060  ON SC=11 TO 20
9070   PRINT "Were your frogs three-legged?"
9080  ON SC=21 TO 40
9090   PRINT "Not bad for a shortsighted toad"
9100  ON SC=41 TO 50
9110   PRINT "Pretty good considering..."
9120  ON SC=51 TO 60
9130   PRINT "A great fly eating operation"
9140  ON SC=REMAINDER
9150   PRINT "Try entering the frog olympics"
9160 END SELect
9170 AT 15,5
9180 PRINT "You scored ";SC
9190 REPeat an_game
9200  INPUT " Another game (Y/N)?";A$
```

```
9210  IF A$="Y" OR A$="N" THEN EXIT an_game
9220 END REPeat an_game
9230 IF A$="Y" THEN
9240  AGAIN=TRUE
9250 ELSE
9260  AGAIN=FALSE
9270 END IF
9299 END DEFine end_game
```

Evaluation and improvements

In its current form 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. The game is fun to play and there is clearly plenty of room for improvement by way of sound effects and extra animation, but the main problem with the game is that it only exercises one sort of skill and this can become boring. The main challenge of the game as currently written is that it is difficult to position the seesaw under the falling frog, and this provides the game's initial interest. Once you have learned how to control the seesaw there is very little 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 improve on this.

If you think about it for a moment, you will see that 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 frog that lands on it. This means that if a frog moving to the left of the screen lands on the seesaw the frog that leaves it continues to move to the left. However, it is not difficult to realise that a frog that is catapulted off a left-hand seesaw should move to the right and one that is catapulted off a right-hand seesaw should move to the left, no matter which way the incoming frog is moving. It is quite easy to make this change to the game by adding the following line to *seesaw*:

```
4525 IF M=1 THEN
4526  B=ABS(B)
4527 ELSE
4528  B=-ABS(B)
4529 END IF
```

This certainly alters how the frog bounces off the seesaw in a way that 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 switch between the two versions of the seesaw by pressing the up arrow key:

```
3035 IF A=4 THEN flip

4700 DEFine PROCedure flip
4710 M=NOT M
4720 PAUSE 10
4799 END DEFine flip
```

The time delay in *flip* (line 4720) 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 player 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 to eat most flies. There are many more possibilities for using the seesaw to control the way in which 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 is left to your imagination.

The final version - a complete listing

```
 10  REMark Leap Frog
 20  WINDOW 512,256,0,0
 25  title
 30  init_graph
 35  REPeat game
 36   init
 50   print_flies
 60   startup
 70   REPeat main
 80    move_ss
 90    move_f
100   IF GAME_END THEN EXIT main
```

```
110  END REPeat main
120  end_game
130  IF NOT AGAIN THEN EXIT game
140 END REPeat game
199 STOP

350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430  READ B
440  POKE_W I,B
450 END FOR I
510 C_TAB=RESPR(32*9)
520 def_graph 0,2,6,130,84,56,186,254,130,68,255
530 def_graph 1,2,6,1,2,4,24,56,88,152,24
540 def_graph 2,2,6,252,0,0,0,0,0,0,0
550 def_graph 3,1,6,36,24,255,255,126,60,102,66
560 def_graph 4,4,6,130,84,56,186,254,130,68,130
570 def_graph 5,6,6,255,255,255,255,255,255,255,255
575 def_graph 6,2,2,255,255,255,255,255,255,255,255
576 def_graph 7,2,6,128,64,32,24,28,26,25,24
577 def_graph 8,2,6,63,0,0,0,0,0,0,0
580 FALSE=0
590 TRUE=1
599 END DEFine init_graph

1000 DEFine PROCedure init
1010 M=1
1030 X%=23
1040 FROG=1
1050 GAME_END=FALSE
1060 AGAIN=FALSE
1070 SC=0
1099 END DEFine init

1300 DEFine PROCedure def_graph(S%,F%,B%,
     R0%,R1%,R2%,R3%,R4%,R5%,R6%,R7%)
1310 LOCal A,FH%,FL%,BH%,BL%
1320 A=C_TAB+32*S%
1330 FH%=(F% DIV 4)*2
1340 FL%=F% && 3
1350 BH%=(B% DIV 4)*2
1360 BL%=B% && 3
1370 charline(R0%)
1380 charline(R1%)
```

```
1390 charline(R2%)
1400 charline(R3%)
1410 charline(R4%)
1420 charline(R5%)
1430 charline(R6%)
1440 charline(R7%)
1499 END DEFine def_graph

1500 DEFine PROCedure charline(R%)
1502 LOCal M%,J,I,DL%,DH%
1503 M%=128
1504 FOR J=1 TO 2
1505  DL%=0:DH%=0
1510  FOR I=1 TO 4
1520   IF (R% && M%)=M% THEN
1530    DL%=FH%+DL%*4
1540    DH%=FL%+DH%*4
1550   ELSE
1560    DL%=BH%+DL%*4
1570    DH%=BL%+DH%*4
1580   END IF
1590   M%=M% DIV 2
1600  END FOR I
1610  POKE A,DL%
1620  POKE A+1,DH%
1630  A=A+2
1640 END FOR J
1699 END DEFine charline

1700 DEFine PROCedure graph(X%,Y%,S%)
1710 CALL do_graph,X%,Y%,S%,C_TAB
1799 END DEFine graph

2000 DEFine PROCedure print_flies
2010 CLS
2020 C%=0
2030 FOR V=2 TO 10 STEP 2
2040  C%=NOT C%
2050  FOR U=2 TO 30 STEP 2
2060   graph U-C%,V,3
2070  END FOR U
2080 END FOR V
2099 END DEFine print_flies

3000 DEFine PROCedure move_ss
3005 LOCal A
3010 A=KEYROW(1)
3020 IF A=16 AND X%<28 THEN X%=X%+1
3030 IF A=2 AND X%>0 THEN X%=X%-1
3035 IF A=4 THEN flip
```

```
3040 print_ss
3099 END DEFine move_ss

3100 DEFine PROCedure print_ss
3110 SELect ON M
3120  ON M=1
3130   graph X%,21,5
3140   graph X%+1,21,0
3150   graph X%+2,21,1
3180   graph X%+3,21,2
3220   graph X%+4,21,5
3240  ON M=0
3250   graph X%,21,5
3260   graph X%+1,21,8
3270   graph X%+2,21,7
3280   graph X%+3,21,0
3290   graph X%+4,21,5
3360 END SELect
3999 END DEFine print_ss

4000 DEFine PROCedure move_f
4010 graph F%,G%,5
4020 F=F+B
4030 G=G+A
4040 A=A+P
4050 IF F>30 OR F<2 THEN B=-B
4060 IF G<2 THEN G=2:A=-A
4070 G%=G
4080 F%=F
4090 IF G%<11 THEN hit_fly
4100 graph F%,G%,4
4110 IF G%>20 THEN seesaw
4199 END DEFine move_f

4500 DEFine PROCedure seesaw
4510 D%=F-X%-M
4520 IF D%<1 OR D%>2 THEN miss:END DEFine seesaw
4525 IF M=1 THEN
4526  B=ABS(B)
4527 ELSE
4528  B=-ABS(B)
4529 END IF
4530 M=NOT M
4540 A=-A
4550 G=20
4560 F=M*3+X%
4570 G%=G
4575 F%=F
4580 BEEP .1,10
4590 PAUSE 5
```

```
4600 BEEP
4699 END DEFine seesaw

4700 DEFine PROCedure flip
4710 M=NOT M
4720 PAUSE 10
4799 END DEFine flip

5000 DEFine PROCedure startup
5010 move_ss
5020 FOR I=26 TO 31
5030  graph I,15,6
5040 END FOR I
5050 FOR F=31 TO 24 STEP -1
5060  graph F+1,14,5
5070  graph F,14,4
5080  move_ss
5090 END FOR F
5100 FOR G=14 TO 19
5110  graph 24,G-1,5
5120  graph 24,G,4
5130  move_ss
5140 END FOR G
5145 graph 24,19,5
5150 F%=24
5155 F=F%
5160 G%=19
5165 G=G%
5170 A=.5+RND/10
5180 B=-(.2+RND/2.5)
5190 P=9E-3
5200 AT 21,2
5210 PRINT "FROG ";FROG;" SCORE=";SC
5220 A=A+A/D
5230 B=B+B/D
5240 seesaw
5299 END DEFine start_up

6000 DEFine PROCedure title
6010 PAPER 6
6015 INK 1
6020 CLS
6030 AT 5,10
6040 PRINT "L E A P  F R O G"
6050 AT 10,4
6060 PRINT "In this game you must eat the flies"\\
6070 PRINT TO 3;"by jumping your two frogs off the
     "\\
6080 PRINT TO 3; "seesaw using the left and right
     arrow"\\
```

```
6090 PRINT TO 3; "keys - you have ten frogs to
     clear"\\
6100 PRINT TO 3;"the screen ....."
6110 REPeat diff
6120  AT 21,5
6130  PRINT "Which difficulty level "
6140  PRINT TO 8;"1 (difficult) to 10 (easy)";
6150  INPUT D
6160  IF D>0 AND D<11 THEN EXIT diff
6170 END REPeat diff
6199 END DEFine title

7000 DEFine PROCedure hit_fly
7010 IF pix(G%,F%,4,4)<>1 THEN END DEFine hit_fly
7020 BEEP .1,20
7030 PAUSE 2
7040 BEEP
7050 SC=SC+1
7060 AT 21,2
7070 PRINT "FROG ";FROG;" SCORE=";SC
7080 IF SC=76 THEN GAME_END=TRUE
7099 END DEFine hit_fly

8000 DEFine PROCedure miss
8010 graph F%,G%,5
8020 FROG=FROG+1
8030 IF FROG<10 THEN
8040  BEEP .1,-10
8050  PAUSE 6
8060  BEEP
8070  M=1
8080  startup
8090 ELSE
8100  GAME_END=TRUE
8110 END IF
8199 END DEFine miss

9000 DEFine PROCedure end_game
9010 CLS
9020 AT 10,3
9030 SELect ON SC
9040  ON SC=0 TO 10
9050   PRINT "A mere tadpole could do better"
9060  ON SC=11 TO 20
9070   PRINT "Were your frogs three-legged?"
9080  ON SC=21 TO 40
9090   PRINT "Not bad for a shortsighted toad"
9100  ON SC=41 TO 50
9110   PRINT "Pretty good considering..."
9120  ON SC=51 TO 60
```

```
9130    PRINT "A great fly eating operation"
9140   ON SC=REMAINDER
9150    PRINT "Try entering the frog olympics"
9160 END SELect
9170 AT 15,5
9180 PRINT "You scored ";SC
9190 REPeat an_game
9200 INPUT " Another game (Y/N)?";A$
9210 IF A$="Y" OR A$="N" THEN EXIT an_game
9220 END REPeat an_game
9230 IF A$="Y" THEN
9240  AGAIN=TRUE
9250 ELSE
9260  AGAIN=FALSE
9270 END IF
9299 END DEFine end_game

9500 DEFine FuNction pix(R%,C%,X%,Y%)
9510 LOCal A%,P%,B%,A
9520 P%=C%*8+X%
9530 A=131072+(512*R%+Y%*64+P% DIV 4)*2
9540 B%=PEEK(A+1)
9550 A%=PEEK(A)
9560 P%=P% MOD 4
9570 IF P%=3 THEN RETurn ((2 && A%)*2)+(3 && B%)
9580 IF P%=2 THEN RETurn ((8 && A%)DIV 2)+
     ((12 && B%) DIV 4)
9590 IF P%=1 THEN RETurn ((32 && A%)DIV 8)+
     ((48 && B%) DIV 16)
9600 IF P%=0 THEN RETurn ((128 && A%) DIV 32)+
     ((192 && B%) DIV 64)
9699 END DEFine pix
```

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 be moved up by one line, so 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 a way of moving a large number of characters in a very short time and with very little programming effort. For example, try:

```
10 MODE 8
20 AT 19,31: PRINT
30 PRINT TO RND(1 TO 31);"*";
40 GO TO 20
```

This program prints asterisks at random positions at the bottom of the screen and animates them vertically up the screen using scrolls. As this program demonstrates, it is possible to animate a large number of objects using scrolling animation. In fact the problem in scrolling

animation is keeping things still! This can be done by printing objects at the end of one scroll and 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 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

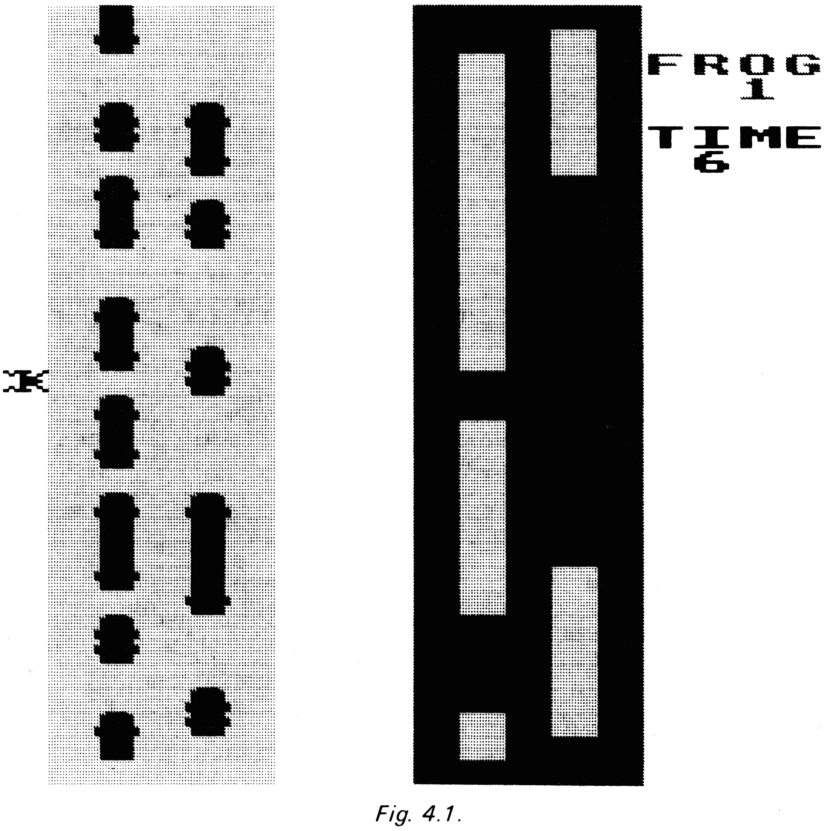


Fig. 4.1.

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, a time limit is imposed. Also, while the player can move the frog up and down the screen and rightwards 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 and then immediately onto the second log.

If the lanes of traffic move in opposite directions and the lines of logs move at different rates then the game becomes much more interesting. Trying to judge the gaps is much harder when the traffic moves both ways. It is also difficult to decide when to cross a river in which logs move at different speeds. Suppose the second line of logs moves twice as fast as the first. In this case any overlap between logs in the two moving lines is only temporary – logs in the second line continually overtake those in the first and the frog can jump onto a log only to find that the log that he was about to jump on next has floated upstream!

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 sounds like a difficult problem but in fact it is very easy using the QL's text window facility. Using its channels 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-line text window for each lane of traffic and each set of logs and scroll them independently. For example, try typing:

```
OPEN #4,scr_20x256a50x0
```

and then type:

```
LIST #4
```

You will find that following this command all the output is confined to a single column on the screen. After you have seen this demonstration of a single column text window it is not difficult to imagine how it could be used to animate the traffic and logs in Frogling.

Main program and procedure *title*

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  REMark frogling
 20  title
 30  init_graph
 40  REPeat main
 50   FOR FROG=1 TO 3
 60    init
 70    AT 2,32:PRINT "FROG"
 80    AT 3,35:PRINT FROG
 90    SEC=0
100    SDATE 2000,1,1,0,0,0
110    REPeat jump
120     scrll
130     get_time
140     move_f
150     IF SEC-5>0 THEN
160      AT 5,34:PRINT "TIME"
170      AT 6,35:PRINT SEC
180     END IF
190     IF SEC-10>MAX OR GAME_END THEN EXIT jump
200    END REPeat jump
210    death
220   END FOR FROG
```

```
230   end_game
240   IF AGAIN=FALSE THEN EXIT main
250  END REPeat main
299  STOP
```

First, a call is made to procedure *title* which simply prints the directions for the game. Then *init_graph* is called to set up the user-defined graphics characters. The rest of the main program is in the form of three nested loops. Lines 40 to 250 form a REPeat loop that plays the game. Lines 50 to 220 contain a FOR loop that repeats the main body of the game three times with three different frogs. The innermost loop, lines 110 to 200, is the animation loop. Procedure *init*, called at line 60, does the usual job of initialising everything that needs to be initialised. Lines 70 and 80 print the number of the current frog before the game proper begins. The animation loop calls procedure *scrll* to animate the road and the river and after obtaining the time using *get_time* in line 130, then calls procedure *move_f* to allow the player to move the frog. Finally, lines 150 to 180 print the current time and line 190 checks for the end of the game. Procedure *death* 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 procedure *end_game* which also asks if another game is desired. The only other point worth mentioning is the use of SEC-10 in the EXIT statement at line 190. This is because for the first ten seconds that the animation loop is running the frog cannot move. This ten-second startup 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 ten seconds and then the frog appears and the timer starts.

Procedure *title* is so simple that it is listed below without comment:

```
8000 DEFine PROCedure title
8010 PAPER 0
8020 INK 4
8030 WINDOW 512,256,0,0
8040 CLS
8050 AT 2,10:PRINT "F R O G L I N G"
8060 AT 5,9: PRINT "In this game you"\
8070 PRINT TO 11;"have to guide"\
8080 PRINT TO 9;"three frogs across"\
8090 PRINT TO 10;"a busy road and a"\
8100 PRINT TO 9;"fast flowing river."\\
8110 PRINT TO 9;"Use the up, down"\
8120 PRINT TO 11;"and the right"\
8130 PRINT TO 12;"arrow keys"\\
```

```
8140 PRINT TO 9;"Don't go under a"\
8150 PRINT TO 10;"car and don't"\
8160 PRINT TO 9;"fall in the river."\\
8170 PRINT TO 11;"Press any key"
8180 PRINT TO 13;"to start"
8190 A$=INKEY$(-1)
8192 AT 22,12
8194 FLASH 1
8196 PRINT "PLEASE WAIT"
8198 FLASH 0
8199 END DEFine title
```

Procedures *init_graph, init* and *get_time*

Procedure *init_graph* operates in the usual way, defining the graphics characters and storing the machine code used by *graph* (remember to refer to Chapter One for details of this procedure and also *charline*, *def_graph* and the function *pix*).

```
350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430 READ B
440 POKE_W I,B
450 END FOR I
500 RANDOMISE
510 C_TAB=RESPR(32*10)
520 def_graph 0,1,0,255,255,255,255,255,255,255,255
530 def_graph 1,2,0,255,255,255,255,255,255,255,255
540 def_graph 2,0,0,255,255,255,255,255,255,255,255
550 def_graph 3,1,2,60,126,126,126,126,255,255,255
560 def_graph 4,1,2,126,126,126,126,126,126,126,126
570 def_graph 5,1,2,126,126,255,255,255,126,126,126
580 def_graph 6,2,1,255,255,255,255,255,255,255,255
590 def_graph 7,4,0,185,82,28,30,28,82,185,0
600 def_graph 8,4,1,185,82,28,30,28,82,185,0
610 def_graph 9,4,2,185,82,28,30,28,82,185,0
620 AGAIN=0
699 END DEFine init_graph
```

Lines 520 to 610 define the cars, logs, frog and necessary spaces (see Table 4.1). As usual, different characters have to be defined to represent the same shape on each of the different background colours.

Table 4.1. Graphics characters

Number	Shape	Foreground	Background
0	space	blue	black
1	space	red	black
2	space	black	black
3	car front	blue	red
4	lorry middle	blue	red
5	car back	blue	red
6	log	red	blue
7	frog	green	black
8	frog	green	blue
9	frog	green	red

You can see the way that the cars and lorries are built up in Fig. 4.2. You may be surprised at the choice of blue cars on a red road and red

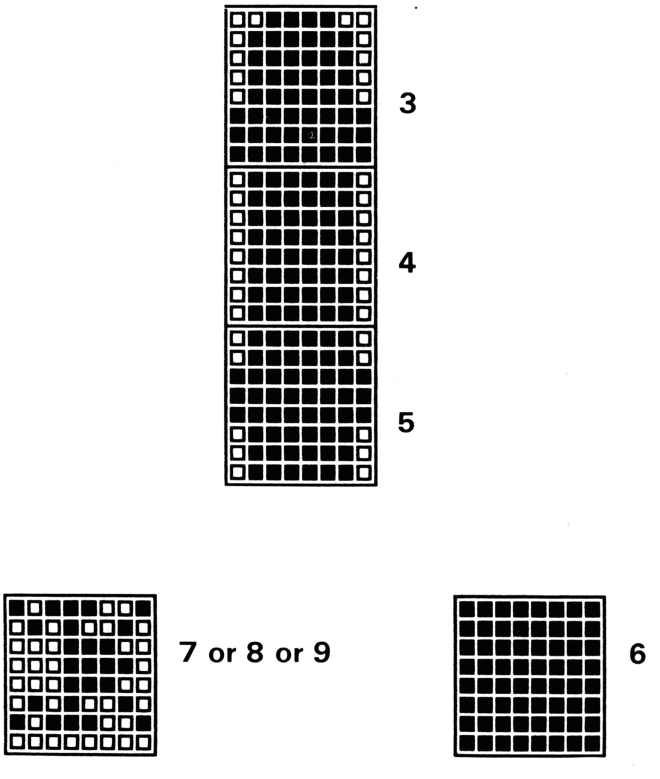


Fig. 4.2. Graphics characters for lorry, frog and log.

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.

Procedure *init* then defines the variables, and in this case is also responsible for printing the coloured strips that represent the road and the river and defining the necessary windows for scrolling the logs and traffic.

```
1000 DEFine PROCedure init
1010 A%=1
1020 B%=A%
1030 C%=0
1040 D%=C%
1050 F%=8
1055 CH%=2
1060 XF=5
1070 YF=15
1075 CLS
1080 FOR K=0 TO 29
1090  FOR J=7 TO 12
1100   graph J,K,1
1110   graph J+9,K,0
1120  END FOR J
1130 END FOR K
1140 FALSE=0
1150 TRUE=1
1160 GAME_END=FALSE
1170 MAX=50
1180 OPEN #3, scr_16x240a128x0
1190 OPEN #4, scr_16x240a176x0
1200 OPEN #5, scr_16x240a272x0
1210 OPEN #6, scr_16x240a320x0
1220 PAPER #3,2
1230 PAPER #4,2
1240 PAPER #5,1
1260 PAPER #6,1
1299 END DEFine init
```

Lines 1010 and 1020 initialise the variables A% and B% to 1 – the code for a red space. Similarly, lines 1030 and 1040 initialise C% and D% to 0 – the code for a blue space. These variables are used by procedure *scrll* and are best described in that section. Lines 1050 to 1070 initialise four variables associated with the frog shape. F% is set to 8 and thus holds a frog on a black background, i.e. the frog as it appears at the start of the game. CH% is set to 2 and is used for the space to blank the frog. XF and YF are the current co-ordinates of the frog and are initialised by lines 1060 and 1070.

Lines 1080 to 1130 print two bands of colour down the screen. The red band to the left-hand side of the screen represents the road and the blue band to the right represents the river. Finally, line 1160 sets the 'GAME_END' flag to FALSE and line 1170 sets the time limit for the game.

Procedure *get_time* gets the time as a string from the QL's DATE$ function and splits it in order to obtain the time in seconds since the start of the game:

```
300 DEFine PROCedure get_time
310 T$=DATE$
320 SEC=T$(17)*60
330 SEC=SEC+T$(19 TO 20)
339 END DEFine get_time
```

Procedure *scrll* and its 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 *scrll* and the procedures that it calls and this is the subject of this section:

```
3000 DEFine PROCedure scrll
3010 A%=gen_car (.2,.5,A%,1)
3020 p_wind 1,A%
3030 B%=gen_car (.16,.5,B%,2)
3040 p_wind 2,B%
3050 C%=gen_log (.16,.2,C%)
3060 p_wind 3,C%
3090 D%=gen_log (.16,.2,C%)
3100 p_wind 4,C%
3110 D%=gen_log (.1,.2,D%)
3120 p_wind 4,D%
3199 END DEFine scrll
```

At first sight *scrll* looks complicated but this is only because the purpose of the functions *gen_car* and *gen_log* and the procedure *p_wind* are 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 *gen_car* (PSTART,PEND,OB%,N%) 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 on the road. PEND is the probability that a car or lorry will come to an 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 code of the last user-defined graphics character that was printed in the lane. For example, if the last character in lane one was a space (code 1) 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. N% determines which of the four windows is to be scrolled.

In *scrll* A% is used to hold the character printed in traffic lane one and line 3010 calls *gen_car* to update it. Function *gen_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. Function *gen_log* performs the same action as *gen_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 *gen_car* and *gen_log* will become clearer after they have been described in more detail later.

Procedure *p_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 code of the character to be printed.

You should now be able to understand the structure of *scrll*. Line 3010 calls *gen_car* to generate the character to be printed in the first lane of traffic. Then line 3020 calls *p_wind* to print it in window one. In the same way lines 3030 and 3040 generate and print a car character for window two. Lines 3050 and 3060 generate and print a log character in text window three and lines 3090 to 3120 generate and print two log characters in text window four. As text window four is scrolled twice each time text window three is scrolled, the second set of logs moves twice as fast as the first set.

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

```
2000 DEFine FuNction gen_car(PSTART,PEND,OB%,N%)
2010 LOCal T%,F%,B%
2015 IF N%=2 THEN
2016  F%=3
2017  B%=5
2018 ELSE
2019  F%=5
2020  B%=3
2021 END IF
2025 T%=OB%
2030 IF OB%=1 AND RND<PSTART THEN T%=F%
2040 IF OB%=F% THEN T%=4
2050 IF OB%=B% THEN T%=1
2060 IF T%=4 AND RND<PEND THEN T%=B%
2070 RETurn T%
2099 END DEFine gen_car
```

Lines 2015 to 2020 place the correct characters in the variables holding the start and end characters. This takes care of the fact that lane one moves in the opposite direction to lane two. Line 2030 tests whether the last character printed was a space. 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 2040 makes the next character the middle section of a lorry. Line 2050 checks to see if the last character was a car end and if it was, the next character will be a blank. Finally, line 2060 changes lorry middle sections to car ends at random.

Function *gen_log* is simpler than *gen_car*:

```
5000 DEFine FuNction gen_log(PSTART,PEND,OB%)
5010 IF OB%=0 AND RND<PSTART THEN RETurn 6
5020 IF OB%=6 AND RND<PEND THEN RETurn 0
5030 RETurn OB%
5099 END DEFine gen_log
```

If the last character was a space then line 5010 will make the next character a log with probability PSTART. Line 5020 does the reverse and changes a log character into a space with probability PEND. You should be able to see that PSTART is the probability of starting a log and PEND is the probability of finishing a log.

Procedure *p_wind* scrolls text windows and prints the next character:

```
4000 DEFine PROCedure p_wind (N%,C%)
4010 SELect ON N%
4020  ON N%=1
4030   SCROLL #3,8
4040   graph 8,0,C%
```

```
4050  ON N%=2
4060   SCROLL #4,-8
4070   graph 11,29,C%
4080  ON N%=3
4090   SCROLL #5,-8
4100   graph 17,29,C%
4110  ON N%=4
4120   SCROLL #6,-8
4130   graph 20,29,C%
4140 END SELect
4199 END DEFine p_wind
```

The first window scrolls in the opposite direction to all the others so if N% is 1, line 4020 scrolls text window one downwards and line 4040 prints the appropriate character at the top of the screen. Lines 4060, 4090 and 4120 scroll text windows two, three and four respectively. These three windows are scrolled up the screen and their relevant characters are printed at the bottom of the screen by lines 4070, 4100 and 4130.

Procedures *move_f, change_char* and *off_s*

On the face of it *move_f* has a very simple job to do and should be similar to all other procedures that move objects around the screen. However, *move_f* has a number of difficulties to overcome. In particular, the character 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. This problem is taken care of by calling another procedure, *change_char*, which is responsible for deciding which version of the frog and space characters should be used according to the horizontal screen position. It also works out where a space should be printed to blank out the frog at its old position.

```
6000 DEFine PROCedure change_char
6010 SELect ON XF
6020  ON XF=1 TO 7
6030   CH%=2
6040   F%=7
6050   K%=0
6060  ON XF=8
6070   CH%=1
6080   F%=9
6090   K%=-1
6100  ON XF=11
6110   CH%=1
```

```
6120    F%=9
6130    K%=1
6140   ON XF=12 TO 15
6150    CH%=2
6160    F%=7
6170    K%=0
6180   ON XF=16 TO 20
6190    CH%=1
6200    F%=9
6210    K%=0
6220   ON XF=21 TO 26
6230    CH%=2
6240    F%=7
6250    K%=0
6260 END SELect
6499 END DEFine change_char
```

The way in which the frog moves depends on whether it is within one of the scrolling text windows or not, and this is the principal remaining task of procedure *move_f*. For example, if it is within the first text window (i.e. the first lane of the road) it will be scrolled down (and if in the second, 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 fourth text windows then, as long as it is sitting on a log, it should be scrolled upward along with the rest of the logs.

```
6500 DEFine PROCedure move_f
6505 IF SEC<10 THEN END DEFine move_f
6510 change_char
6520 graph XF,YF-K%,CH%
6530 I%=KEYROW(1)
6540 IF I%=4 AND YF>2 THEN YF=YF-1
6550 IF I%=128 AND YF<31 THEN YF=YF+1
6560 IF I%=16 THEN
6570  BEEP .1,-10:PAUSE 2:BEEP
6580  XF=XF+3
6590 END IF
6600 Z%=pix(YF,XF,4,4)
6610 IF Z%=1 THEN
6620  F%=8
6630  GAME_END=TRUE
6640  graph XF,YF,F%
6645 ELSE
6650 change_char
6660 graph XF,YF,F%
6665 END IF
6670 SELect ON XF
```

```
6680  ON XF=17
6690   off_s
6710  ON XF=20
6720   off_s
6721   off_s
6730  ON XF=18 TO 23
6740   GAME_END=TRUE
6750 END SELect
6799 END DEFine move_f
```

Line 6505 stops the frog from moving until ten seconds have elapsed since the start of the game. Line 6502 prints the correct blanking space at the correct position. Lines 6530 to 6590 alter the frog's co-ordinates depending on which arrow key is pressed.

Line 6600 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 1, then line 6630 sets GAME_END to TRUE. Otherwise line 6650 calls *change_char* to select a character of the correct background colour and then line 6660 prints the frog.

Lines 6670 to 6750 test to see if the frog is in either of text windows three and four and if this is the case procedure *off_s* 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 6721 calls *off_s* a second time if the frog is in text window four to allow for the fact that text window four is scrolled twice. Finally, line 6730 tests to see if the frog has safely reached the far side of the river.

It only remains to give the listing of *off_s*; remember that *pix* is as listed in Chapter One.

```
7200 DEFine PROCedure off_s
7210 IF YF=0 THEN
7220  GAME_END=TRUE
7230 ELSE
7250  YF=YF-1
7260 END IF
7299 END DEFine off_s
```

Procedures *death* and *end_game*

The final procedures in the game, *death* and *end_game*, are very simple. Procedure *death* tests to find out whether the frog made its journey successfully and if not, how it met its end, and sounds a note accordingly. Procedure *end_game* asks the player if another game is required.

```
7500 DEFine PROCedure death
7510 IF XF>20 THEN
7520  BEEP .1,10
7530  PAUSE 5
7540  BEEP
7550 ELSE
7560  BEEP .1,5
7570  PAUSE 2
7580  BEEP
7590 END IF
7600 PAUSE 50
7699 END DEFine death

8500 DEFine PROCedure end_game
8510 CLS
8520 REPeat inp
8530 AT 10,3
8540 PRINT "Another game Y/N ?";
8550 INPUT A$
8560 IF A$(1)="Y" OR A$(1)="N" THEN EXIT inp
8570 END REPeat inp
8580 IF A$(1)="Y" THEN AGAIN=TRUE
8599 END DEFine end_game
```

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 a rather disappointing aspect of the game is its dull sound effects. This is soon put to rights with the addition of a call to a new procedure, *ribbit*, which in turn calls another procedure, *rib*, to produce a sound with a rapidly rising and falling pitch reminiscent of a frog croaking.

```
9800 DEFine PROCedure ribbit
9810 rib
9820 BEEP
9830 PAUSE 2
9840 rib
9850 BEEP
9899 END DEFine ribbit

9900 DEFine PROCedure rib
9910 BEEP .1,20,5,0,7,2
9920 PAUSE 6
9999 END DEFine rib
```

These procedures can be used to make a more appropriate sound when the frog jumps. Change line 6570 to:

```
6570  ribbit
```

The addition of this simple sound effect certainly increases both the tension and the enjoyment of the game and you should try playing the game both with and without it. Even though this is the point at which we left Frogling, this doesn't mean that there isn't 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 procedure *end_game* to print appropriate comments along the lines of those found in Leap Frog.

The final version - a complete listing

```
 10 REMark frogling
 20 title
 30 init_graph
 40 REPeat main
 50  FOR FROG=1 TO 3
 60   init
 70   AT 2,32:PRINT "FROG"
 80   AT 3,35:PRINT FROG
 90   SEC=0
100   SDATE 2000,1,1,0,0,0
110   REPeat jump
120    scrll
130    get_time
140    move_f
150    IF SEC-5>0 THEN
160     AT 5,34:PRINT "TIME"
170     AT 6,35:PRINT SEC
180    END IF
190    IF SEC-10>MAX OR GAME_END THEN EXIT jump
200   END REPeat jump
210   death
220  END FOR FROG
230  end_game
240  IF AGAIN=FALSE THEN EXIT main
250 END REPeat main
299 STOP

300 DEFine PROCedure get_time
310 T$=DATE$
320 SEC=T$(17)*60
330 SEC=SEC+T$(19 TO 20)
339 END DEFine get_time
350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
```

```
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430 READ B
440 POKE_W I,B
450 END FOR I
500 RANDOMISE
510 C_TAB=RESPR(32*10)
520 def_graph 0,1,0,255,255,255,255,255,255,255,255
530 def_graph 1,2,0,255,255,255,255,255,255,255,255
540 def_graph 2,0,0,255,255,255,255,255,255,255,255
550 def_graph 3,1,2,60,126,126,126,126,255,255,255
560 def_graph 4,1,2,126,126,126,126,126,126,126,126
570 def_graph 5,1,2,126,126,255,255,255,126,126,126
580 def_graph 6,2,1,255,255,255,255,255,255,255,255
590 def_graph 7,4,0,185,82,28,30,28,82,185,0
600 def_graph 8,4,1,185,82,28,30,28,82,185,0
610 def_graph 9,4,2,185,82,28,30,28,82,185,0
620 AGAIN=0
699 END DEFine init_graph

1000 DEFine PROCedure init
1010 A%=1
1020 B%=A%
1030 C%=0
1040 D%=C%
1050 F%=8
1055 CH%=2
1060 XF=5
1070 YF=15
1075 CLS
1080 FOR K=0 TO 29
1090  FOR J=7 TO 12
1100   graph J,K,1
1110   graph J+9,K,0
1120  END FOR J
1130 END FOR K
1140 FALSE=0
1150 TRUE=1
1160 GAME_END=FALSE
1170 MAX=50
1180 OPEN #3, scr_16x240a128x0
1190 OPEN #4, scr_16x240a176x0
1200 OPEN #5, scr_16x240a272x0
1210 OPEN #6, scr_16x240a320x0
1220 PAPER #3,2
1230 PAPER #4,2
```

```
1240 PAPER 5,1
1260 PAPER 6,1
1299 END DEFine init

1300 DEFine PROCedure def_graph(S%,F%,B%,
     R0%,R1%,R2%,R3%,R4%,R5%,R6%,R7%)
1310 LOCal A,FH%,FL%,BH%,BL%
1320 A=C_TAB+32*S%
1330 FH%=(F% DIV 4)*2
1340 FL%=F% && 3
1350 BH%=(B% DIV 4)*2
1360 BL%=B% && 3
1370 charline(R0%)
1380 charline(R1%)
1390 charline(R2%)
1400 charline(R3%)
1410 charline(R4%)
1420 charline(R5%)
1430 charline(R6%)
1440 charline(R7%)
1499 END DEFine def_graph

1500 DEFine PROCedure charline(R%)
1510 LOCal M%,J,I,DL%,DH%
1520 M%=128
1530 FOR J=1 TO 2
1540  DL%=0:DH%=0
1550  FOR I=1 TO 4
1560   IF (R% && M%)=M% THEN
1570    DL%=FH%+DL%*4
1580    DH%=FL%+DH%*4
1590   ELSE
1600    DL%=BH%+DL%*4
1610    DH%=BL%+DH%*4
1620   END IF
1630   M%=M% DIV 2
1640  END FOR I
1650  POKE A,DL%
1660  POKE A+1,DH%
1670  A=A+2
1680 END FOR J
1699 END DEFine charline

1700 DEFine PROCedure graph(X%,Y%,S%)
1710 CALL do_graph,X%,Y%,S%,C_TAB
1799 END DEFine graph

2000 DEFine FuNction gen_car(PSTART,PEND,OB%,N%)
2010 LOCal T%,F%,B%
2015 IF N%=2 THEN
```

```
2016  F%=3
2017  B%=5
2018 ELSE
2019  F%=5
2020  B%=3
2021 END IF
2025 T%=OB%
2030 IF OB%=1 AND RND<PSTART THEN T%=F%
2040 IF OB%=F% THEN T%=4
2050 IF OB%=B% THEN T%=1
2060 IF T%=4 AND RND<PEND THEN T%=B%
2070 RETurn T%
2099 END DEFine gen_car

3000 DEFine PROCedure scrll
3010 A%=gen_car (.2,.5,A%,1)
3020 p_wind 1,A%
3030 B%=gen_car (.16,.5,B%,2)
3040 p_wind 2,B%
3050 C%=gen_log (.16,.2,C%)
3060 p_wind 3,C%
3090 D%=gen_log (.16,.2,C%)
3100 p_wind 4,C%
3110 D%=gen_log (.1,.2,D%)
3120 p_wind 4,D%
3199 END DEFine scrll

4000 DEFine PROCedure p_wind (N%,C%)
4010 SELect ON N%
4020  ON N%=1
4030   SCROLL #3,8
4040   graph 8,0,C%
4050  ON N%=2
4060   SCROLL #4,-8
4070   graph 11,29,C%
4080  ON N%=3
4090   SCROLL #5,-8
4100   graph 17,29,C%
4110  ON N%=4
4120   SCROLL #6,-8
4130   graph 20,29,C%
4140 END SELect
4199 END DEFine p_wind

5000 DEFine FuNction gen_log(PSTART,PEND,OB%)
5010 IF OB%=0 AND RND<PSTART THEN RETurn 6
5020 IF OB%=6 AND RND<PEND THEN RETurn 0
5030 RETurn OB%
5099 END DEFine gen_log
```

```
6000 DEFine PROCedure change_char
6010 SELect ON XF
6020  ON XF=1 TO 7
6030   CH%=2
6040   F%=7
6050   K%=0
6060  ON XF=8
6070   CH%=1
6080   F%=9
6090   K%=-1
6100  ON XF=11
6110   CH%=1
6120   F%=9
6130   K%=1
6140  ON XF=12 TO 15
6150   CH%=2
6160   F%=7
6170   K%=0
6180  ON XF=16 TO 20
6190   CH%=1
6200   F%=9
6210   K%=0
6220  ON XF=21 TO 26
6230   CH%=2
6240   F%=7
6250   K%=0
6260 END SELect
6499 END DEFine change_char

6500 DEFine PROCedure move_f
6505 IF SEC<10 THEN END DEFine move_f
6510 change_char
6520 graph XF,YF-K%,CH%
6530 I%=KEYROW(1)
6540 IF I%=4 AND YF>2 THEN YF=YF-1
6550 IF I%=128 AND YF<31 THEN YF=YF+1
6560 IF I%=16 THEN
6570  ribbit
6580  XF=XF+3
6590 END IF
6600 Z%=pix(YF,XF,4,4)
6610 IF Z%=1 THEN
6620  F%=8
6630  GAME_END=TRUE
6640  graph XF,YF,F%
6645 ELSE
6650 change_char
6660 graph XF,YF,F%
6665 END IF
6670 SELect ON XF
```

```
6680  ON XF=17
6690   off_s
6710  ON XF=20
6720   off_s
6721   off_s
6730  ON XF=18 TO 23
6740   GAME_END=TRUE
6750 END SELect
6799 END DEFine move_f

7200 DEFine PROCedure off_s
7210 IF YF=0 THEN
7220  GAME_END=TRUE
7230 ELSE
7250  YF=YF-1
7260 END IF
7299 END DEFine off_s

7500 DEFine PROCedure death
7510 IF XF>20 THEN
7520  BEEP .1,10
7530  PAUSE 5
7540  BEEP
7550 ELSE
7560  BEEP .1,5
7570  PAUSE 2
7580  BEEP
7590 END IF
7600 PAUSE 50
7699 END DEFine death

8000 DEFine PROCedure title
8010 PAPER 0
8020 INK 4
8030 WINDOW 512,256,0,0
8040 CLS
8050 AT 2,10:PRINT "F R O G L I N G"
8060 AT 5,9: PRINT "In this game you"\
8070 PRINT TO 11;"have to guide"\
8080 PRINT TO 9;"three frogs across"\
8090 PRINT TO 10;"a busy road and a"\
8100 PRINT TO 9;"fast flowing river."\\
8110 PRINT TO 9;"Use the up, down"\
8120 PRINT TO 11;"and the right"\
8130 PRINT TO 12;"arrow keys"\\
8140 PRINT TO 9;"Don't go under a"\
8150 PRINT TO 10;"car and don't"\
8160 PRINT TO 9;"fall in the river."\\
8170 PRINT TO 11;"Press any key"
8180 PRINT TO 13;"to start"
```

```
8190 A$=INKEY$(-1)
8192 AT 22,12
8194 FLASH 1
8196 PRINT "PLEASE WAIT"
8198 FLASH 0
8199 END DEFine title

8500 DEFine PROCedure end_game
8510 CLS
8520 REPeat inp
8530 AT 10,3
8540 PRINT "Another game Y/N ?";
8550 INPUT A$
8560 IF A$(1)="Y" OR A$(1)="N" THEN EXIT inp
8570 END REPeat inp
8580 IF A$(1)="Y" THEN AGAIN=TRUE
8599 END DEFine end_game

9500 DEFine FuNction pix(R%,C%,X%,Y%)
9510 LOCal A%,P%,B%,A
9520 P%=C%*8+X%
9530 A=131072+(512*R%+Y%*64+P% DIV 4)*2
9540 B%=PEEK(A+1)
9550 A%=PEEK(A)
9560 P%=P% MOD 4
9570 IF P%=3 THEN RETurn ((2 && A%)*2)+(3 && B%)
9580 IF P%=2 THEN RETurn ((8 && A%)DIV 2)+
     ((12 && B%) DIV 4)
9590 IF P%=1 THEN RETurn ((32 && A%)DIV 8)+
     ((48 && B%) DIV 16)
9600 IF P%=0 THEN RETurn ((128 && A%) DIV 32)+
     ((192 && B%) DIV 64)
9699 END DEFine pix

9800 DEFine PROCedure ribbit
9810 rib
9820 BEEP
9830 PAUSE 2
9840 rib
9850 BEEP
9899 END DEFine ribbit

9900 DEFine PROCedure rib
9910 BEEP .1,20,5,0,7,2
9920 PAUSE 6
9999 END DEFine rib
```

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 of movement of the head. 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 character moves 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 was 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 out 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 in addition the old position of the tail has to be blanked out because there isn't a 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 8:WINDOW 512,256,0,0
 15 CLS
 20 XH=10:YH=15
 30 XT=0:YT=15
 40 AT YH,XH:PRINT "S";
 50 AT YT,XT:PRINT " ";
 60 XH=XH+1
 70 XT=XT+1
 80 IF XH>40 THEN XH=1
 90 IF XT>40 THEN XT=1
100 PAUSE 5
110 GO TO 40
```

Line 20 defines the position of the head at XH,YH and line 30 sets the

old position of the tail to be at XT,YT. Line 40 prints the head at its new position and line 50 blanks out 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 be 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 the problem of 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* 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 head changes its direction it is altogether different. Each time the snake moves, the tail moves to the position that was occupied by the last-but-one character and so it is clearly necessary always to know the position of the last-but-one character. This argument can be repeated because each time the snake moves, the second-from-last character 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)
END FOR 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 of 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:

```
graph X(N),Y(N),H%
```

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

```
graph X(N-1),Y(N-1),S%
```

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

```
graph XT,YT,3
graph X(1),Y(1),T%
```

where H%, S% and T% are strings containing the character used for the head, the body and the tail of the snake respectively and graphics character 3 is the space of the correct colour.

The only trouble with the above method is that each time the snake moves the contents of the pair of arrays X(I) and Y(I) 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 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 *The Complete Programmer*, by Mike James (Granada, 1983). 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 moving 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 are 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 co-ordinates 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).

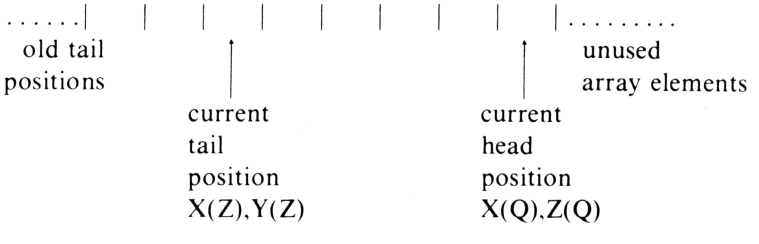


Fig. 5.1.

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 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 1. 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 now 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 character longer and so a little more difficult and exciting to control. To increase the difficulty of the game still further the snake must not only avoid running into itself but must 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.

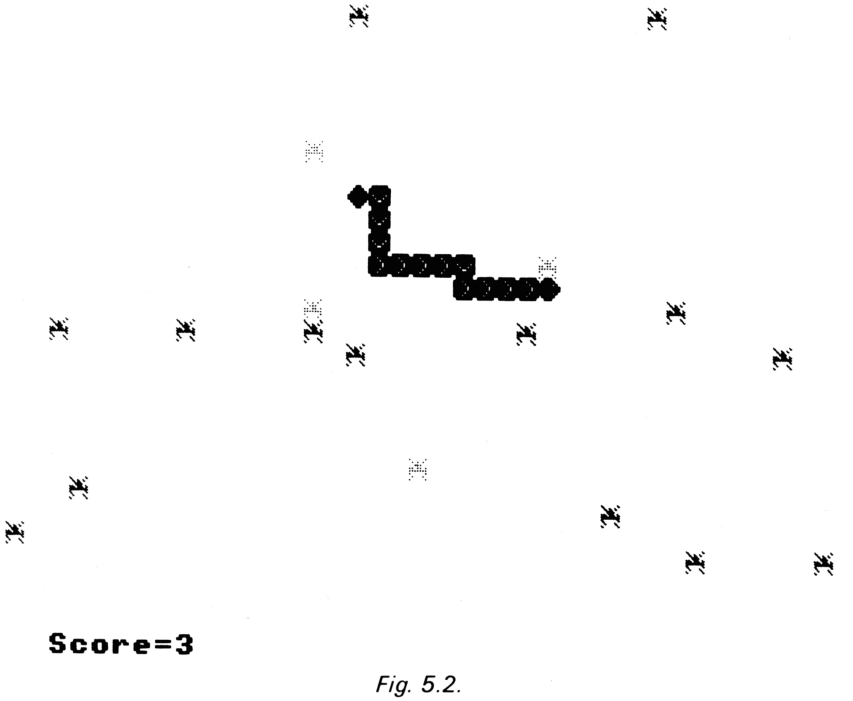


Fig. 5.2.

The main program

Four colours are used in this game; we chose to use yellow for the background, black and yellow for the snake, green for the frogs and red for the poisonous toads. Once again a window of 512 by 256 is defined to give sufficient space for a fairly large snake to manoeuvre. The main program for this game follows the usual form:

```
 10  REMark SNAKE
 20  WINDOW 512,256,0,0
 30  init_graph
 40  title
 50  REPeat game
 60   init
 70   draw_scene
 80   REPeat main
 90    one_move
100   IF NO_FOOD OR NO_LIVES THEN EXIT main
110  END REPeat main
120  end_game
130  IF NOT AGAIN THEN EXIT game
140 END REPeat game
199 STOP
```

Line 30 calls *init_graph* which initialises the graphics characters and sets AGAIN to 0. 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 130 tests AGAIN and exits the loop back if no further game has been requested. Line 40 calls procedure *title* to display the title frame. Lines 50 to 140 form the repeat loop responsible for playing the game. Line 60 calls procedure *init* to initialise the variables used in the game and then line 70 calls *draw_scene* to draw the frogs and toads and the snake in its starting position.

Lines 80 to 110 form the animation loop. Procedure *one_move* is responsible for moving the snake and checking to see if it has hit anything. Line 100 tests for the end of the game. The variable NO_FOOD is true if the snake has managed to eat all the frogs without using up its five lives and NO_LIVES is true if the snake has either run into itself or into a poisonous toad five times and hence used up its lives. Finally procedure *end_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.

Procedures *init_graph* and *init*

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 – graphics characters 0, 3, 7 and 9 (one for each direction of travel) – there are also four different 'snake body' characters – 1, 4, 8 and 10 (see Fig. 5.3). By selecting a body character

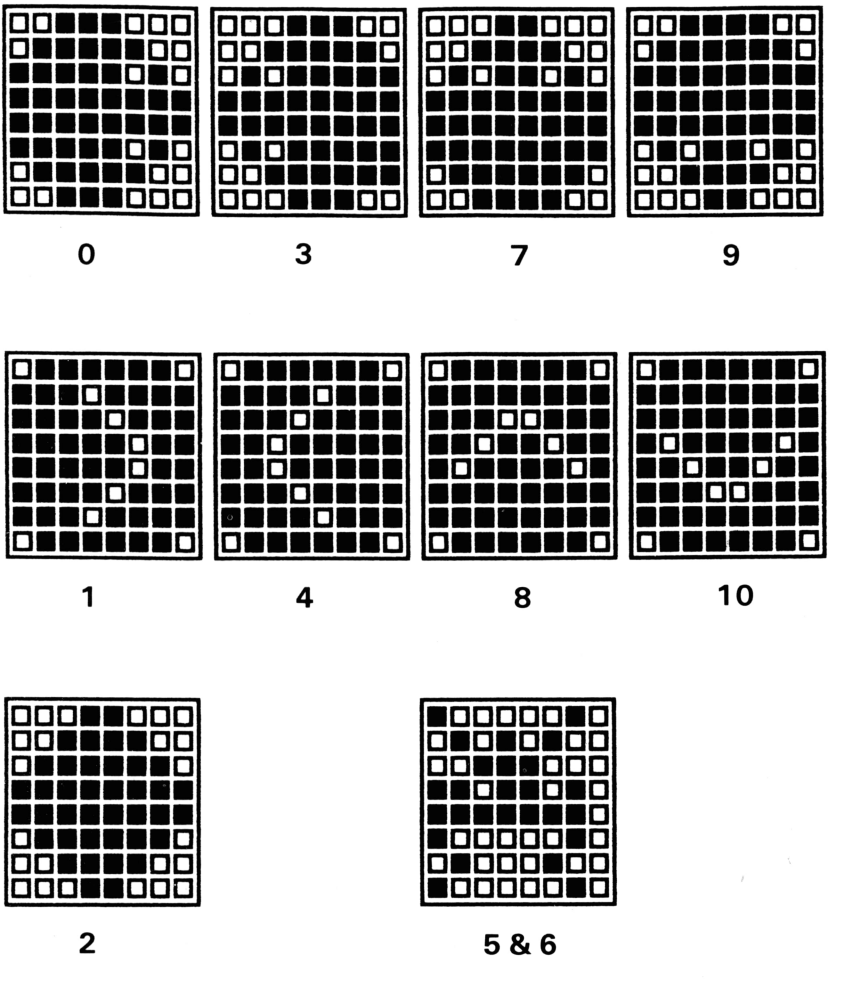


Fig. 5.3. Graphics characters for snake's head, snake's body, snake's tail and frog.

that points in the same direction as the head of the snake to overprint the head at each move, the snake can be made to look even more impressive and full of movement. There is only one character (2) used

for the tail because the diamond shape looks the same in every direction of travel. The frog, which has foreground colour green, is defined as character 5 and the poisonous toad, which is the same shape but with foreground colour red, is defined as character 6. Character 1 is a yellow space which is used to blank out the snake. Table 5.1 summarises the graphics characters:

Table 5.1. Graphics characters

Number	Shape	Foreground	Background
0	snake head right	black	yellow
1	snake body right	black	yellow
2	snake tail	black	yellow
3	snake head left	black	yellow
4	snake body left	black	yellow
5	frog	green	yellow
6	toad	red	yellow
7	snake head up	black	yellow
8	snake body up	black	yellow
9	snake head down	black	yellow
10	snake body down	black	yellow
11	space	yellow	yellow

Procedure *init_graph* has a format that should be becoming familiar to you:

```
350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430 READ B
440 POKE_W I,B
450 END FOR I
510 C_TAB=RESPR(32*12)
520 def_graph 0,0,6,56,124,250,255,255,250,124,56
530 def_graph 1,0,6,126,239,247,251,251,247,239,126
540 def_graph 2,0,6,24,60,126,255,255,126,60,24
550 def_graph 3,0,6,28,62,95,255,255,95,62,28
560 def_graph 4,0,6,126,247,239,223,223,239,247,126
570 def_graph 5,4,6,130,84,56,60,254,130,68,130
580 def_graph 6,2,6,130,84,56,60,254,130,68,130
```

```
590 def_graph 7,0,6,24,60,90,255,255,255,126,60
600 def_graph 8,0,6,60,255,231,219,189,255,255,126
610 def_graph 9,0,6,60,126,255,255,126,90,60,24
620 def_graph 10,0,6,126,255,255,189,219,231,255,126
630 def_graph 11,6,6,255,255,255,255,255,255,255,255
640 TRUE=1
650 FALSE=0
660 AGAIN=TRUE
670 DIM X%(50)
680 DIM Y%(50)
690 MAX%=50
700 DIM T$(5,15)
710 DIM T(5)
720 FOR I=1 TO 5
730  T(I)=0
740 END FOR I
799 END DEFine init_graph
1000 DEFine PROCedure init
1010 NO_FOOD=FALSE
1020 NO_LIVES=FALSE
1030 HIT=FALSE
1040 C%=0
1050 STOTAL%=0
1060 SC=0
1070 SN=0
1080 FOOD=0
1099 END DEFine init

1300 DEFine PROCedure def_graph(S%,F%,B%,
     R0%,R1%,R2%,R3%,R4%,R5%,R6%,R7%)
1310 LOCal A,FH%,FL%,BH%,BL%
1320 A=C_TAB+32*S%
1330 FH%=(F% DIV 4)*2
1340 FL%=F% && 3
1350 BH%=(B% DIV 4)*2
1360 BL%=B% && 3
1370 charline(R0%)
1380 charline(R1%)
1390 charline(R2%)
1400 charline(R3%)
1410 charline(R4%)
1420 charline(R5%)
1430 charline(R6%)
1440 charline(R7%)
1499 END DEFine def_graph
```

The first part of *init_graph* (lines 360 to 450) again stores the machine code used by *graph* in a reserved area of memory as described in Chapter One. User-defined characters are defined using the machine code described in previous chapters. Lines 510 to 630

then define the twelve graphics characters described above. Lines 640 and 650 set the values of TRUE and FALSE and 660 sets the initial value of AGAIN to be TRUE – this means the game will run at least once, as explained in the previous section. Lines 670 and 680 set up two arrays, X% and Y%, that will be used later in the program to store the snake's co-ordinates. MAX% (line 690) 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 700 and 710, are used later in the program to hold a league table of highest scores and lines 720 to 740 set the initial values stored in the table to be 0.

Procedure *init* sets the end of game indicators to FALSE (lines 1010 to 1030) and zeros the counters and scores used. 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 *title* is fairly simple, apart from its use of the snake characters to write the word 'SNAKE' in large letters across the screen.

```
8000 DEFine PROCedure title
8010 PAPER 6
8020 CLS
8030 INK 0
8040 snake
8045 CLS
8050 AT 4,2
8060 PRINT "You must guide your snake using"
8065 PRINT "  the four arrow keys and"
8070 PRINT "  eat all the green frogs";
8080 graph 19,8,5
8085 PRINT \\
8090 PRINT "  If you try to eat a toad "
8100 graph 20,10,6
8110 PRINT "  or yourself"
```

```
8120 PRINT "  then you will lose one of"
8130 PRINT "  your five lives"\\
8140 PRINT "  You get five screens"
8145 PRINT "  of frogs to eat before you"
8150 PRINT "  become a SUPER SNAKE"
8155 REPeat wait
8156  AT 23,7
8160  FLASH 1
8170  PRINT "PRESS ANY KEY"
8180  FLASH 0
8190  IF INKEY$<>"" THEN EXIT wait
8195 END REPeat wait
8199 END DEFine title
```

Lines 8080 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 procedure *snake*:

```
8500 DEFine PROCedure snake
8510 DATA 8,1,1,0,11,8,1,1,1,11,8,1,1,1,11
8520 DATA 2,11,8,0,11,8,1,1,0
8530 DATA 8,11,11,11,11,8,11,11,10,11,8,11,11,10
8540 DATA 11,10,8,1,11,11,8,11,11,11
8545 DATA 4,4,4,8,11,8,11,11,10,11,8,1,1
8550 DATA 10,11,10,8,11,11,11,4,4,8,11
8560 DATA 11,11,11,8,11,8,11,11,10,11,8,10,1,1
8570 DATA 11,10,4,8,11,11,8,1,1,11
8580 DATA 11,11,11,8,11,8,11,11,10,11,8,11,11
8590 DATA 10,11,10,11,4,8,11,8,11,11,11
8600 DATA 2,1,1,1,11,2,11,11,9,11,2,11,11,9
8610 DATA 11,9,11,11,2,11,4,4,4,2
8620 RESTORE 8500
8630 FOR I=9 TO 14
8640  FOR J=3 TO 26
8650   READ S
8660   graph J,I,S
8670  END FOR J
8680 END FOR I
8690 PAUSE 100
8699 END DEFine snake
```

The DATA statements in lines 8510 to 8610 are simply lists indicating characters to be printed. The DATA statements are organised in pairs and each pair defines a single row of 23 characters. Six 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 them using a range of different standard size characters.

Procedures *draw_scene, draw_snake* and *drawn_food*

Although it isn't obvious at this stage in the program design, *draw_scene* is in fact used by other procedures within the program to re-draw the scene after 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 for it to 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 reset it to a random value, so generating a new screenful of frogs.

```
2000 DEFine PROCedure draw_scene
2010 CLS
2020 draw_snake
2030 draw_food 6,4*SC
2040 IF FOOD=0 THEN FOOD=RND(1 TO 10)+15
2050 draw_food 5,FOOD
2060 AT 22,3
2070 PRINT "Score=";STOTAL%+C%
2099 END DEFine draw_scene
```

The call to *draw_snake* produces a fixed length snake at the same starting position each time. Procedure *draw_snake* is also responsible for initialising the co-ordinate arrays X% and Y% and the pointers Q% and Z%. The procedure *draw_food* will produce both frogs and toads depending on the value of the variable F passed to it from *draw_scene*. Line 2030 uses it to print toads and line 2050 uses it to print frogs. The number of toads increases as each screenful of frogs is eaten, so 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 2040 generates a random number of frogs to fill yet another screen. Finally, line 2070 prints the total score.

Procedure *draw_snake* looks a little complicated because it not only draws the snake, it also initialises the co-ordinate arrays X% and Y%:

```
2600 DEFine PROCedure draw_snake
2610 A%=2
2620 B%=10
2630 XH%=A%
2640 YH%=B%
2650 H%=0
2660 S%=1
2670 T%=2
```

```
2680 graph A%,B%,T%
2690 X%(1)=A%
2700 Y%(1)=B%
2710 FOR Q=2 TO 10
2730  A%=A%+1
2740  graph A%,B%,S%
2750  X%(Q)=A%
2760  Y%(Q)=B%
2770 END FOR Q
2780 A%=A%+1
2790 graph A%,B%,H%
2800 X%(11)=A%
2810 Y%(11)=B%
2820 XV%=1
2830 YV%=0
2840 Z%=1
2850 Q%=11
2899 END DEFine draw_snake
```

The variables A%,B% 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 procedure *update*, 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 2680 prints the tail character at 2,10 and then lines 2690 and 2700 store its co-ordinates in X%(1) and Y%(1). The FOR loop, lines 2710 to 2770, prints the characters that make up the body of the snake and stores their co-ordinates in X% and Y%. Finally, lines 2780 and 2790 print the head and then lines 2800 to 2850 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 XV% and YV% control the direction of movement and are explained in more detail as part of procedures *get_direction* and *update*.

Procedure *draw_food* is very simple:

```
2500 DEFine PROCedure draw_food (F,AMOUNT)
2510 IF AMOUNT=0 THEN END DEFine draw_food
2520 FOR K=1 TO AMOUNT
2525  REPeat pos
2530   U%=RND(1 TO 30)
2540   V%=RND(1 TO 21)
2550   IF pix(V%,U%,1,1)=6 THEN EXIT pos
2560  END REPeat pos
2570  graph U%,V%,F
2580 END FOR K
2599 END DEFine draw_food
```

The FOR loop, lines 2520 to 2580, will print AMOUNT frogs or toads at random locations. Whether the procedure produces frogs or toads depends on the value of F. The loop from line 2525 to line 2560 generates random numbers to be used as the potential co-ordinates of a frog or toad. If there is no character at V%,U% then line 2550 exits the loop; otherwise, another two random numbers are generated. The test for an existing character is performed by way of the function *pix*, which returns the colour of a pixel at the top left of the character location at V%,U%. (*Pix* was described in Chapter One and its listing is given there.)

Procedure *one_move* and its associated procedures

Procedure *one_move* simply calls two other procedures to do all the work necessary to move the snake:

```
3000 DEFine PROCedure one_move
3010 get_direction
3020 update
3099 END DEFine one_move
```

Procedure *get_direction* examines the keyboard to see whether an 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 procedure *update*. 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.

Procedure *get_direction* is very similar to the other keyboard routines used in earlier programs:

```
4000 DEFine PROCedure get_direction
4002 BEEP .1,-20
4004 PAUSE 2
4006 BEEP
4010 A=KEYROW(1)
4020 IF A=0 THEN END DEFine get_direction
4030 SELect ON A
4040  ON A=4
4050   XV%=0
4060   YV%=-1
4070   H%=7
4080   S%=8
4090  ON A=128
4100   XV%=0
```

```
4110    YV%=1
4120    H%=9
4130    S%=10
4140   ON A=2
4150    XV%=-1
4160    YV%=0
4170    H%=3
4180    S%=4
4190   ON A=16
4200    XV%=1
4210    YV%=0
4220    H%=0
4230    S%=1
4240 END SELect
4299 END DEFine get_direction
```

Lines 4002 to 4006 make a sort of hissing noise using the BEEP command. The KEYROW function is then used to check each arrow key in turn (lines 4010 to 4240). 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 XV% and YV% are added to A% and B% (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 4220 sets H% to a right-facing snake's head and line 4230 sets S% to a right-pointing body character.

Procedure *update* is probably the most complicated procedure in the entire program:

```
4500 DEFine PROCedure update
4510 XH%=A%:YH%=B%
4520 A%=A%+XV%:B%=B%+YV%
4530 IF A%>30 THEN A%=1
4540 IF A%<1 THEN A%=30
4550 IF B%>25 THEN B%=0
4560 IF B%<0 THEN B%=25
4570 IF pix(B%,A%,4,4)<>6 THEN hit_f
4580 graph XH%,YH%,S%
4590 graph A%,B%,H%
4600 Q%=Q%+1
4610 IF Q%>MAX% THEN Q%=1
4620 X%(Q%)=A%:Y%(Q%)=B%
4630 graph X%(Z%),Y%(Z%),11
4640 Z%=Z%+1
```

```
4650 IF Z%>MAX% THEN Z%=1
4660 graph X%(Z%),Y%(Z%),T%
4699 END DEFine update
```

Line 4510 saves the current position of the head, given by A%,B% in XH%,YH%. Then line 4520 updates the position of the head by adding the velocities XV% and YV%. Following this, lines 4530 to 4560 make sure that the new position is still on the screen. There 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 *pix* to check if the snake has hit anything. If it has, then *hit_f* is called to deal with the situation. Otherwise line 4580 prints a body character at the head's old position so as to blank it out and line 4590 prints the head at its new position. Notice that procedure *update* doesn't have to worry about which of the many head or body characters to use because the correct ones have been selected by procedure *get_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 procedure *update* is to move the snake's tail. Line 4630 blanks out the tail as its old position. Lines 4640 and 4650 then update the tail pointer, Z%, and line 4600 prints the tail character at its new position.

Procedure *hit_f* and its associated procedures

Procedure *hit_f* is responsible for producing the actions that are required whenever the snake runs into anything:

```
5000 DEFine PROCedure hit_f
5010 COL=pix(B%,A%,4,4)
5020 SELect ON COL
5030  ON COL=4
5040   frog
5050  ON COL=REMAINDER
5060   toad_self
5070  END SELect
5099 END DEFine hit_f
```

When *hit_f* is called, all that is known is that the snake has run into

something and the first thing that *hit_f* has to do is discover what! Line 5010 uses *pix* 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 red or black then the snake has hit a toad or itself respectively and *toad_self* is called (line 5060) to take away one of the snake's lives. However, if the colour turns out to be green then *frog* is called to add 1 to the snake's score and make the snake one character longer (line 5040).

Procedure *toad_self* is quite short but the way it works and interacts with the rest of the program is quite complicated.

```
5500 DEFine PROCedure toad_self
5510 BEEP .1,-10
5512 PAUSE 2
5515 BEEP
5520 BEEP .1,-20
5524 PAUSE 2
5526 BEEP
5530 SN=SN+1
5540 IF SN<4 THEN
5560  draw_scene
5570 ELSE
5580  STOTAL%=STOTAL%+C%
5590  NO_LIVES=TRUE
5600 END IF
5699 END DEFine toad_self
```

Lines 5510 to 5526 make a noise to let the player know that a life has been lost. Then line 5530 adds 1 to the snake counter, SN. If SN is greater than 4 then all the lives have been used up and the game is over. Line 5540 tests for this and line 5590 sets NO_LIVES to TRUE to inform the main program that this is the case. Otherwise *draw_scene* is called to re-draw the snake and the remaining frogs and toads ready for the game to continue.

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

```
5700 DEFine PROCedure frog
5710 BEEP .1,10
5715 PAUSE 2
5720 BEEP
5730 C%=C%+1
5740 FOOD=FOOD-1
5750 AT 22,3
5760 PRINT "Score=";STOTAL%+C%
5770 Z%=Z%-1
5780 IF Z%<1 THEN Z%=MAX%
```

```
5790 IF FOOD<>0 THEN END DEFine frog
5800 STOTAL%=STOTAL%+C%
5810 C%=0
5820 IF SC<4 THEN
5830  SC=SC+1
5840  draw_scene
5850 ELSE
5860  NO_FOOD=TRUE
5870 END IF
5899 END DEFine frog
```

Lines 5710 to 5720 make a noise to indicate to the player that a frog has just been eaten. Then line 5730 adds 1 to the current score and line 5740 subtracts 1 from FOOD. The reason why FOOD has to be reduced by 1 is that it is used to test whether all the frogs have been eaten and as an indicator of how many frogs should be redrawn if the snake loses a life and the game is restarted. Lines 5750 and 5760 print the total score.

Increasing the length of the snake by one character is simplicity itself. Lines 5770 and 5780 adjust the tail pointer Z%, so moving the current location of the tail back by one position. When *frog* ends, control is returned to *update* 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 *frog* is concerned with detecting whether all the frogs have been eaten. If this is the case, line 5800 adds C% to the total. Line 5820 then tests to see if all five screenfuls of frogs have been eaten and line 5860 sets NO_FOOD to TRUE if this has happened. If there are more screens of frogs still to eat then the scene number, SC, is incremented (line 5830) and *draw_scene* is called (line 5840) to draw a new screenful of frogs. Notice that *draw_scene* will detect the fact that FOOD is zero and so generate a new random number of frogs.

Procedure *end_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 DEFine PROCedure end_game
7010 CLS
7015 AT 3,10
7020 SELect ON SC
7030  ON SC=0
7040   PRINT "Demoted to slowworm"
7050  ON SC=1
7060   PRINT "The frogs don't have to worry"
7070  ON SC=2
7080   PRINT "Not bad for a grass snake"
7090  ON SC=3
7100   PRINT "Well slithered"
7110  ON SC=4
7140   IF NO_FOOD THEN
7150    super
7160    snake
7170   ELSE
7180    PRINT "A venomous performance"
7190   END IF
7200 END SELect
7210 FOR I= 5 TO 1 STEP -1
7220  IF STOTAL%>=T(I) THEN N=I
7230 END FOR I
7235 AT 15,10
7240 IF N=0 THEN
7250  PRINT "You scored ";STOTAL%
7260 ELSE
7270  PRINT "You are now ranked ";N
7280  PRINT TO 10;"in the snake league"
7290 IF N<5 THEN
7300  FOR I=5 TO N+1 STEP -1
7310   T(I)=T(I-1)
7320   T$(I)=T$(I-1)
7330  END FOR I
7340 END IF
7350 T(N)=STOTAL%
7355 PRINT TO 10;
7360 INPUT "What is your name ";T$(N)
7370 CLS
7380 FOR I=1 TO 5
7390  AT I*2+5,5
7400  PRINT T$(I,1 TO 15);
7410  PRINT TO 21; T(I)
7420 END FOR I
7440 REPeat an_game
7445  AT 21,10
7450  INPUT "Another slither ";A$
7455  IF A$="Y" OR A$="N" THEN EXIT an_game
7460  END REPeat an_game
7470 IF A$="N" THEN
```

```
7480  AGAIN=FALSE
7490 ELSE
7500  AGAIN=TRUE
7510 END IF
7599 END DEFine end_game
```

Lines 7015 to 7200 print the congratulatory messages. If you have successfully cleared all five screenfuls of frogs, then the message 'Super Snake' is generated in large letters made up of snake graphics characters as in the original title frame. This is done by calling the procedures *super* and *snake*. Procedure *super* is very similar to *snake*:

```
6000 DEFine PROCedure super
6010 RESTORE 6010
6020 DATA 8,1,1,0,11,2,11,11,7,11,8,1,1,1
6030 DATA 11,8,1,1,0,11,8,1,1,1
6040 DATA 8,11,11,11,11,10,11,11,8,11,8,11,11,10
6050 DATA 11,8,11,11,11,11,8,11,11,10
6060 DATA 4,4,4,8,11,10,11,11,8,11,8,3,4,10
6070 DATA 11,8,4,8,11,11,8,4,4,10
6080 DATA 11,11,11,8,11,10,11,11,8,11,8,11,11,11
6090 DATA 11,8,1,1,11,11,8,10,1,11
6092 DATA 11,11,11,8,11,10,11,11,8,11,8,11,11,11
6094 DATA 11,8,11,11,11,11,8,11,10,1
6100 DATA 2,1,1,1,11,10,1,1,1,11,2,11,11,11,11
6110 DATA 4,4,4,2,11,2,11,11,9
6120 PAUSE 20
6130 CLS
6140 FOR I=2 TO 7
6150  FOR J=3 TO 26
6160   READ S%
6170   graph J,I,S%
6180  END FOR J
6190 END FOR I
6199 END DEFine super
```

An additional feature of the end of the snake game is that it maintains a league table of scores (lines 7290 to 7420). 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 7210 to 7230 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 7290 to 7330) and your score and name are inserted (lines 7350 to 7360). Then the league table is printed (lines 7370 to 7420) and the player is asked if another game is required (lines 7440 to 7460). Notice that if another game is required it isn't 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 isn't 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 for using the snake in another game will be explored.

The final version - a complete listing

```
 10  REMark SNAKE
 20  WINDOW 512,256,0,0
 30  init_graph
 40  title
 50  REPeat game
 60   init
 70   draw_scene
 80   REPeat main
 90    one_move
100   IF NO_FOOD OR NO_LIVES THEN EXIT main
110  END REPeat main
120  end_game
130  IF NOT AGAIN THEN EXIT game
140 END REPeat game
199 STOP

350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430 READ B
440 POKE_W I,B
450 END FOR I
510 C_TAB=RESPR(32*12)
520 def_graph 0,0,6,56,124,250,255,255,250,124,56
530 def_graph 1,0,6,126,239,247,251,251,247,239,126
```

```
540 def_graph 2,0,6,24,60,126,255,255,126,60,24
550 def_graph 3,0,6,28,62,95,255,255,95,62,28
560 def_graph 4,0,6,126,247,239,223,223,239,247,126
570 def_graph 5,4,6,130,84,56,60,254,130,68,130
580 def_graph 6,2,6,130,84,56,60,254,130,68,130
590 def_graph 7,0,6,24,60,90,255,255,255,126,60
600 def_graph 8,0,6,60,255,231,219,189,255,255,126
610 def_graph 9,0,6,60,126,255,255,126,90,60,24
620 def_graph 10,0,6,126,255,255,189,219,231,255,126
630 def_graph 11,6,6,255,255,255,255,255,255,255,255
640 TRUE=1
650 FALSE=0
660 AGAIN=TRUE
670 DIM X%(50)
680 DIM Y%(50)
690 MAX%=50
700 DIM T$(5,15)
710 DIM T(5)
720 FOR I=1 TO 5
730  T(I)=0
740 END FOR I
799 END DEFine init_graph

1000 DEFine PROCedure init
1010 NO_FOOD=FALSE
1020 NO_LIVES=FALSE
1030 HIT=FALSE
1040 C%=0
1050 STOTAL%=0
1060 SC=0
1070 SN=0
1080 FOOD=0
1099 END DEFine init

1300 DEFine PROCedure def_graph(S%,F%,B%,
     R0%,R1%,R2%,R3%,R4%,R5%,R6%,R7%)
1310 LOCal A,FH%,FL%,BH%,BL%
1320 A=C_TAB+32*S%
1330 FH%=(F% DIV 4)*2
1340 FL%=F% && 3
1350 BH%=(B% DIV 4)*2
1360 BL%=B% && 3
1370 charline(R0%)
1380 charline(R1%)
1390 charline(R2%)
1400 charline(R3%)
1410 charline(R4%)
1420 charline(R5%)
1430 charline(R6%)
1440 charline(R7%)
```

```
1499 END DEFine def_graph

1500 DEFine PROCedure charline(R%)
1510 LOCal M%,J,I,DL%,DH%
1520 M%=128
1530 FOR J=1 TO 2
1540  DL%=0:DH%=0
1550  FOR I=1 TO 4
1560   IF (R% && M%)=M% THEN
1570    DL%=FH%+DL%*4
1580    DH%=FL%+DH%*4
1590   ELSE
1600    DL%=BH%+DL%*4
1610    DH%=BL%+DH%*4
1620   END IF
1630   M%=M% DIV 2
1640  END FOR I
1650  POKE A,DL%
1660  POKE A+1,DH%
1670  A=A+2
1680 END FOR J
1699 END DEFine charline

1700 DEFine PROCedure graph(X%,Y%,S%)
1710 CALL do_graph,X%,Y%,S%,C_TAB
1799 END DEFine graph

2000 DEFine PROCedure draw_scene
2010 CLS
2020 draw_snake
2030 draw_food 6,4*SC
2040 IF FOOD=0 THEN FOOD=RND(1 TO 10)+15
2050 draw_food 5,FOOD
2060 AT 22,3
2070 PRINT "Score=";STOTAL%+C%
2099 END DEFine draw_scene

2500 DEFine PROCedure draw_food (F,AMOUNT)
2510 IF AMOUNT=0 THEN END DEFine draw_food
2520 FOR K=1 TO AMOUNT
2525  REPeat pos
2530   U%=RND(1 TO 30)
2540   V%=RND(1 TO 21)
2550   IF pix(V%,U%,1,1)=6 THEN EXIT pos
2560  END REPeat pos
2570  graph U%,V%,F
2580 END FOR K
2599 END DEFine draw_food

2600 DEFine PROCedure draw_snake
```

```
2610 A%=2
2620 B%=10
2630 XH%=A%
2640 YH%=B%
2650 H%=0
2660 S%=1
2670 T%=2
2680 graph A%,B%,T%
2690 X%(1)=A%
2700 Y%(1)=B%
2710 FOR Q=2 TO 10
2730  A%=A%+1
2740  graph A%,B%,S%
2750  X%(Q)=A%
2760  Y%(Q)=B%
2770 END FOR Q
2780 A%=A%+1
2790 graph A%,B%,H%
2800 X%(11)=A%
2810 Y%(11)=B%
2820 XV%=1
2830 YV%=0
2840 Z%=1
2850 Q%=11
2899 END DEFine draw_snake

3000 DEFine PROCedure one_move
3010 get_direction
3020 update
3099 END DEFine one_move

4000 DEFine PROCedure get_direction
4002 BEEP .1,-20
4004 PAUSE 2
4006 BEEP
4010 A=KEYROW(1)
4020 IF A=0 THEN END DEFine get_direction
4030 SELect ON A
4040  ON A=4
4050   XV%=0
4060   YV%=-1
4070   H%=7
4080   S%=8
4090  ON A=128
4100   XV%=0
4110   YV%=1
4120   H%=9
4130   S%=10
4140  ON A=2
4150   XV%=-1
```

```
4160    YV%=0
4170    H%=3
4180    S%=4
4190   ON A=16
4200    XV%=1
4210    YV%=0
4220    H%=0
4230    S%=1
4240 END SELect
4299 END DEFine get_direction

4500 DEFine PROCedure update
4510 XH%=A%:YH%=B%
4520 A%=A%+XV%:B%=B%+YV%
4530 IF A%>30 THEN A%=1
4540 IF A%<1 THEN A%=30
4550 IF B%>25 THEN B%=0
4560 IF B%<0 THEN B%=25
4570 IF pix(B%,A%,4,4)<>6 THEN hit_f
4580 graph XH%,YH%,S%
4590 graph A%,B%,H%
4600 Q%=Q%+1
4610 IF Q%>MAX% THEN Q%=1
4620 X%(Q%)=A%:Y%(Q%)=B%
4630 graph X%(Z%),Y%(Z%),11
4640 Z%=Z%+1
4650 IF Z%>MAX% THEN Z%=1
4660 graph X%(Z%),Y%(Z%),T%
4699 END DEFine update

5000 DEFine PROCedure hit_f
5010 COL=pix(B%,A%,4,4)
5020 SELect ON COL
5030  ON COL=4
5040   frog
5050  ON COL=REMAINDER
5060   toad_self
5070  END SELect
5099 END DEFine hit_f

5500 DEFine PROCedure toad_self
5510 BEEP .1,-10
5512 PAUSE 2
5515 BEEP
5520 BEEP .1,-20
5524 PAUSE 2
5526 BEEP
5530 SN=SN+1
5540 IF SN<4 THEN
5560  draw_scene
```

```
5570 ELSE
5580  STOTAL%=STOTAL%+C%
5590  NO_LIVES=TRUE
5600 END IF
5699 END DEFine toad_self

5700 DEFine PROCedure frog
5710 BEEP .1,10
5715 PAUSE 2
5720 BEEP
5730 C%=C%+1
5740 FOOD=FOOD-1
5750 AT 22,3
5760 PRINT "Score=";STOTAL%+C%
5770 Z%=Z%-1
5780 IF Z%<1 THEN Z%=MAX%
5790 IF FOOD<>0 THEN END DEFine frog
5800 STOTAL%=STOTAL%+C%
5810 C%=0
5820 IF SC<4 THEN
5830  SC=SC+1
5840  draw_scene
5850 ELSE
5860  NO_FOOD=TRUE
5870 END IF
5899 END DEFine frog

6000 DEFine PROCedure super
6010 RESTORE 6010
6020 DATA 8,1,1,0,11,2,11,11,7,11,8,1,1,1
6030 DATA 11,8,1,1,0,11,8,1,1,1
6040 DATA 8,11,11,11,11,10,11,11,8,11,8,11,11,10
6050 DATA 11,8,11,11,11,11,8,11,11,10
6060 DATA 4,4,4,8,11,10,11,11,8,11,8,3,4,10
6070 DATA 11,8,4,8,11,11,8,4,4,10
6080 DATA 11,11,11,8,11,10,11,11,8,11,8,11,11,11
6090 DATA 11,8,1,1,11,11,8,10,1,11
6092 DATA 11,11,11,8,11,10,11,11,8,11,8,11,11,11
6094 DATA 11,8,11,11,11,11,8,11,10,1
6100 DATA 2,1,1,1,11,10,1,1,1,11,2,11,11,11,11
6110 DATA 4,4,4,2,11,2,11,11,9
6120 PAUSE 20
6130 CLS
6140 FOR I=2 TO 7
6150  FOR J=3 TO 26
6160   READ S%
6170   graph J,I,S%
6180  END FOR J
6190 END FOR I
6199 END DEFine super
```

```
7000 DEFine PROCedure end_game
7010 CLS
7015 AT 3,10
7020 SELect ON SC
7030  ON SC=0
7040   PRINT "Demoted to slowworm"
7050  ON SC=1
7060   PRINT "The frogs don't have to worry"
7070  ON SC=2
7080   PRINT "Not bad for a grass snake"
7090  ON SC=3
7100   PRINT "Well slithered"
7110  ON SC=4
7140   IF NO_FOOD THEN
7150    super
7160    snake
7170   ELSE
7180    PRINT "A venomous performance"
7190   END IF
7200 END SELect
7210 FOR I= 5 TO 1 STEP -1
7220  IF STOTAL%>=T(I) THEN N=I
7230 END FOR I
7235 AT 15,10
7240 IF N=0 THEN
7250  PRINT "You scored ";STOTAL%
7260 ELSE
7270  PRINT "You are now ranked ";N
7280  PRINT TO 10;"in the snake league"
7290 IF N<5 THEN
7300  FOR I=5 TO N+1 STEP -1
7310   T(I)=T(I-1)
7320   T$(I)=T$(I-1)
7330  END FOR I
7340 END IF
7350 T(N)=STOTAL%
7355 PRINT TO 10;
7360 INPUT "What is your name ";T$(N)
7370 CLS
7380 FOR I=1 TO 5
7390  AT I*2+5,5
7400  PRINT T$(I,1 TO 15);
7410  PRINT TO 21; T(I)
7420 END FOR I
7440 REPeat an_game
7445  AT 21,10
7450  INPUT "Another slither ";A$
7455  IF A$="Y" OR A$="N" THEN EXIT an_game
7460  END REPeat an_game
7470 IF A$="N" THEN
```

```
7480  AGAIN=FALSE
7490 ELSE
7500  AGAIN=TRUE
7510 END IF
7599 END DEFine end_game

8000 DEFine PROCedure title
8010 PAPER 6
8020 CLS
8030 INK 0
8040 snake
8045 CLS
8050 AT 4,2
8060 PRINT "You must guide your snake using"
8065 PRINT "  the four arrow keys and"
8070 PRINT "  eat all the green frogs";
8080 graph 19,8,5
8085 PRINT \\
8090 PRINT "  If you try to eat a toad "
8100 graph 20,10,6
8110 PRINT "  or yourself"
8120 PRINT "  then you will lose one of"
8130 PRINT "  your five lives"\\
8140 PRINT "  You get five screens"
8145 PRINT "  of frogs to eat before you"
8150 PRINT "  become a SUPER SNAKE"
8155 REPeat wait
8156  AT 23,7
8160  FLASH 1
8170  PRINT "PRESS ANY KEY"
8180  FLASH 0
8190  IF INKEY$<>"" THEN EXIT wait
8195 END REPeat wait
8199 END DEFine title

8500 DEFine PROCedure snake
8510 DATA 8,1,1,0,11,8,1,1,1,11,8,1,1,1,11
8520 DATA 2,11,8,0,11,8,1,1,0
8530 DATA 8,11,11,11,11,8,11,11,10,11,8,11,11,10
8540 DATA 11,10,8,1,11,11,8,11,11,11
8545 DATA 4,4,4,8,11,8,11,11,10,11,8,1,1
8550 DATA 10,11,10,8,11,11,11,4,4,8,11
8560 DATA 11,11,11,8,11,8,11,11,10,11,8,10,1,1
8570 DATA 11,10,4,8,11,11,8,1,1,11
8580 DATA 11,11,11,8,11,8,11,11,10,11,8,11,11
8590 DATA 10,11,10,11,4,8,11,8,11,11,11
8600 DATA 2,1,1,1,11,2,11,11,9,11,2,11,11,9
8610 DATA 11,9,11,11,2,11,4,4,4,2
8620 RESTORE 8500
8630 FOR I=9 TO 14
```

```
8640  FOR J=3 TO 26
8650   READ S
8660   graph J,I,S
8670  END FOR J
8680 END FOR I
8690 PAUSE 100
8699 END DEFine snake

9500 DEFine FuNction pix(R%,C%,X%,Y%)
9510 LOCal A%,P%,B%,A
9520 P%=C%*8+X%
9530 A=131072+(512*R%+Y%*64+P% DIV 4)*2
9540 B%=PEEK(A+1)
9550 A%=PEEK(A)
9560 P%=P% MOD 4
9570 IF P%=3 THEN RETurn ((2 && A%)*2)+(3 && B%)
9580 IF P%=2 THEN RETurn ((8 && A%)DIV 2)+
     ((12 && B%) DIV 4)
9590 IF P%=1 THEN RETurn ((32 && A%)DIV 8)+
     ((48 && B%) DIV 16)
9600 IF P%=0 THEN RETurn ((128 && A%)DIV 32)+
     ((192 && B%) DIV 64)
9699 END DEFine pix
```

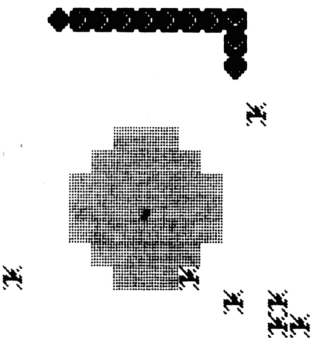
Chapter Six
Tadpole

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.

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 pool in the middle of the screen. One by one 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 pool. 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 pool and the moving frogs, the rest of the elements of Tadpole remain unchanged from the basic Snake game. In particular, the snake has five lives 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. Apart from the tadpole character and a cyan block character for the pool, all the graphics characters used in the new game have already been introduced. The screen layout (see Fig. 6.1) is very similar to that produced by Snake, the only addition being the pool in the middle, but it should be recalled that in Snake the frogs are static, whereas in Tadpole they move.



Snake=1 Screen=1 Score=0

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 to one procedure – *draw_scene*. Now, instead of printing frogs and toads, it has to print a pool containing tadpoles in the middle of the screen. The pool is best produced by way of a new procedure, *draw_pool*, and the tadpoles by a modification to *draw_food*.

The new version of *draw_scene* is:

```
2000 DEFine PROCedure draw_scene
2010 CLS
2020 draw_snake
2030 draw_pool
2040 IF FOOD=0 THEN FOOD=NUM%
2050 draw_food FOOD
2060 AT 24,3
2064 PRINT "Snake=";SN;
2068 PRINT " Screen=";SC;
2070 PRINT " Score=";STOTAL%+C%
```

```
2080 LINE 0,10 TO 256,10
2099 END DEFine draw_scene
```

The new procedure to draw the pool is called at line 2030. Procedure *draw_food* is now called only once (line 2050), to print a fixed number of tadpoles in the pool. The only other changes are the addition of lines 2064 to 2070 to print the snake and screen number, and the inclusion of a high resolution graphics command, line 2080, to draw a horizontal line to show the limit of the playing area.

Procedure *draw_pool* simply uses *graph* to place cyan blocks on the screen in the shape of a pool.

```
2200 DEFine PROCedure draw_pool
2205 LOCal T
2206 T=1
2210 FOR Y=12 TO 14
2220  FOR X=14-T TO 14+T
2230   graph X,Y,12
2240   graph X,Y+(8-(T*2)),12
2250  END FOR X
2255  T=T+1
2260 END FOR Y
2270 FOR X=11 TO 17
2280  graph X,15,12
2290 END FOR X
2299 END DEFine draw_pool
```

All that remains is to define the cyan block as graphics character 12 within *init_graph*:

```
635 def_graph 12,5,5,255,255,255,255,255,255,
    255,255
```

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

Procedure *draw_food* now has to place tadpoles in the pool rather than frogs anywhere on the screen. This is easily catered for by restricting the range of the random numbers generated by lines 2530 and 2540. Later in the program the tadpoles and the frogs that they give rise to have to be animated and this implies that their positions are 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 DEFine PROCedure draw_food (AMOUNT)
2510 IF AMOUNT=0 THEN END DEFine draw_food
2520 FOR K=1 TO AMOUNT
2525  REPeat pos
```

```
2530    U%=RND(1 TO 7)+10
2540    V%=RND(1 TO 7)+9
2550    IF pix(V%,U%,3,3)=5 THEN EXIT pos
2560   END REPeat pos
2570   graph U%,V%,6
2575   FX%(K)=U%:FY%(K)=V%
2580 END FOR K
2582 IF AMOUNT<NUM% THEN
2584  FOR K=AMOUNT+1 TO NUM%
2586    FX%(K)=-1
2588  END FOR K
2590 END IF
2599 END DEFine draw_food
```

Line 2575 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 co-ordinate is set to −1. The FOR loop, lines 2584 to 2588, sets the unused entries of the array FX% to −1. Of course, the new pair of arrays have to be dimensioned and the tadpole character defined, so lines 580, 750 and 760 have to be added to *init_graph*:

```
580 def_graph 6,0,5,0,24,56,24,16,16,32,64

750 NUM%=8
760 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 *move_f* which moves a single frog at random. That is, *one_move* becomes:

```
3000 DEFine PROCedure one_move
3005 IF RND>.5 THEN move_f
3010 get_direction
3020 update
3099 END DEFine one_move
```

Procedure *move_f* is a little more complicated than you might expect. The main problems are 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 DEFine PROCedure move_f
3510 K%=RND(1 TO NUM%)
3520 IF FX%(K%)=-1 THEN  END DEFine move_f
3530 N%=pix(FY%(K%),FX%(K%),4,4)
3540 IF N%=5 THEN
3550  graph FX%(K%),FY%(K%),12
3560 ELSE
3570  IF N%<>0 THEN graph FX%(K%),FY%(K%),11
3580 END IF
3585 REPeat jump
3590  FX%(K%)=FX%(K%)-sgn(13.5-FX%(K%))
3600  FY%(K%)=FY%(K%)-sgn(14.5-FY%(K%))
3610  IF FX%(K%)<0 OR FX%(K%)>31 THEN frog_gone:
      END DEFine move_f
3620  IF FY%(K%)<0 OR FY%(K%)>24 THEN frog_gone:
      END DEFine move_f
3630  N%=pix(FY%(K%),FX%(K%),4,4)
3640  IF N%=6 THEN EXIT jump
3650 END REPeat jump
3660 graph FX%(K%),FY%(K%),5
3699 END DEFine move_f

3700 DEFine FuNction sgn(Q)
3710 IF Q=0 THEN RETurn 0
3720 IF Q>0 THEN
3730  RETurn 1
3740 ELSE
3750  RETurn -1
3760 END IF
3799 END DEFine sgn
```

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 line 3520 returns control to *one_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 lines 3540 to 3580 blank out the frog's old position using the appropriate colour block. Lines 3585 to 3600 update the frog's co-ordinates in such a way that the frog moves toward the edge of the screen. The way this works is easy to understand once you know that *sgn*(13.5−FX%(K%)) is +1 if the frog is to the left of the centre of the pool and −1 if it is to the right of the pool. (Function *sgn* was introduced in Chapter Two.) Thus, subtracting *sgn*(13.5−FX%(K%)) from the current X co-ordinate of the frog always results in it moving

away from the centre of the pool. A similar argument shows that subtracting *sgn*(14.5−FY%(K%)) from the current Y co-ordinate of the frog also results in it moving away from the centre of the pool. Following this, lines 3610 to 3620 test for the frog going off the screen and call *frog_gone* if it has. The final line of the procedure prints the frog at its new position.

Procedure *move_f* calls *frog_gone* to mark the fact that a frog has escaped:

```
3900 DEFine PROCedure frog_gone
3910 FX%(K%)=-1
3920 FOOD=FOOD-1
3930 IF FOOD<>0 THEN END DEFine frog_gone
3940 STOTAL%=STOTAL%+C%
3950 C%=0
3960 IF SC=4 THEN
3962  NO_FOOD=TRUE
3964 ELSE
3970  SC=SC+1
3980  draw_scene
3990 END IF
3999 END DEFine frog_gone
```

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 1 from FOOD and restarts the game with a new screenful of tadpoles/frogs, unless of course five screenfuls have already been eaten or escaped.

Catching frogs

The remaining changes to the program are mainly changes made to *frog*. Procedure *frog* is called whenever the snake runs into a green object, i.e. a frog! In the original version of Snake all *frog* had to do was add 1 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 FY%. This is done by adding a call to a new procedure, *rem_f*:

```
5745 rem_f
```

In addition, the opportunity is taken to add lines 5750 to 5760 to print the snake, screen numbers and score:

```
5750 AT 24,3
```

```
5752 PRINT "Snake=";SN;
5754 PRINT " Screen=";SC;
5760 PRINT " Score=";STOTAL%+C%
```

The position of the frog that the snake has just run into is stored in A% and B%. Procedure *rem_f* removes it from the list of frogs by searching FX% and FY% for a frog with the same co-ordinates, and then setting its X co-ordinate to −1.

```
5900 DEFine PROCedure rem_f
5910 K%=0
5920 REPeat find_f
5930  K%=K%+1
5940  IF A%=FX%(K%) AND B%=FY%(K%) THEN EXIT find_f
5960 END REPeat find_f
5970 FX%(K%)=-1
5999 END DEFine rem_f
```

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

```
7010 CLS
7015 AT 3,10
7016 STOTAL=STOTAL%
7020 SELect ON STOTAL
7030  ON STOTAL=0 TO 9
7040   PRINT "Demoted to slowworm"
7050  ON STOTAL=10 TO 19
7060   PRINT "The frogs don't have to worry"
7070  ON STOTAL=20 TO 29
7080   PRINT "Not bad for a grass snake"
7090  ON STOTAL=30 TO 34
7100   PRINT "Well slithered"
7110  ON STOTAL=REMAINDER
7140   IF NO_FOOD THEN
7150    super
7160    snake
7170   ELSE
7180    PRINT "A venomous performance"
7190   END IF
7200 END SELect
```

and the instructions in *title*:

```
8085 PRINT \\
8090 PRINT " Don't fall in the water or try to"
8110 PRINT "  eat yourself or you will lose"
8130 PRINT "  one of your five lives"\\
8140 PRINT "  You get five screens"
8145 PRINT "  of frogs to eat before you"
8150 PRINT "  become a SUPER SNAKE"
```

Conclusion

The conversion of Snake to Tadpole has been easy and serves to demonstrate the value of using procedures and functions to write a program. There are still many ways in which Tadpole could be improved but the current version of the game is already fun to play and its further development is left for you to explore. You could, for example, allow some of the frogs to be poisonous toads, or allow them to move to avoid the snake ...

The final program - a complete listing

```
 10 REMark TADPOLE
 20 WINDOW 512,256,0,0
 30 init_graph
 40 title
 50 REPeat game
 60  init
 70  draw_scene
 80  REPeat main
 90   one_move
100   IF NO_FOOD OR NO_LIVES THEN EXIT main
110  END REPeat main
120  end_game
130  IF NOT AGAIN THEN EXIT game
140 END REPeat game
199 STOP

350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430  READ B
440  POKE_W I,B
450 END FOR I
510 C_TAB=RESPR(32*13)
520 def_graph 0,0,6,56,124,250,255,255,250,124,56
530 def_graph 1,0,6,126,239,247,251,251,247,239,126
540 def_graph 2,0,6,24,60,126,255,255,126,60,24
550 def_graph 3,0,6,28,62,95,255,255,95,62,28
560 def_graph 4,0,6,126,247,239,223,223,239,247,126
570 def_graph 5,4,6,130,84,56,60,254,130,68,130
```

```
580 def_graph 6,0,5,0,24,56,24,16,16,32,64
590 def_graph 7,0,6,24,60,90,255,255,255,126,60
600 def_graph 8,0,6,60,255,231,219,189,255,255,126
610 def_graph 9,0,6,60,126,255,255,126,90,60,24
620 def_graph 10,0,6,126,255,255,189,219,231,255,126
630 def_graph 11,6,6,255,255,255,255,255,255,255,255
635 def_graph 12,5,5,255,255,255,255,255,255,255,255

640 TRUE=1
650 FALSE=0
660 AGAIN=TRUE
670 DIM X%(50)
680 DIM Y%(50)
690 MAX%=50
700 DIM T$(5,15)
710 DIM T(5)
720 FOR I=1 TO 5
730  T(I)=0
740 END FOR I
750 NUM%=8
760 DIM FX%(NUM%),FY%(NUM%)
799 END DEFine init_graph

1000 DEFine PROCedure init
1010 NO_FOOD=FALSE
1020 NO_LIVES=FALSE
1030 HIT=FALSE
1040 C%=0
1050 STOTAL%=0
1060 SC=0
1070 SN=0
1080 FOOD=0
1099 END DEFine init

1300 DEFine PROCedure def_graph(S%,F%,B%,
     R0%,R1%,R2%,R3%,R4%,R5%,R6%,R7%)
1310 LOCal A,FH%,FL%,BH%,BL%
1320 A=C_TAB+32*S%
1330 FH%=(F% DIV 4)*2
1340 FL%=F% && 3
1350 BH%=(B% DIV 4)*2
1360 BL%=B% && 3
1370 charline(R0%)
1380 charline(R1%)
1390 charline(R2%)
1400 charline(R3%)
1410 charline(R4%)
1420 charline(R5%)
1430 charline(R6%)
1440 charline(R7%)
```

```
1499 END DEFine def_graph

1500 DEFine PROCedure charline(R%)
1510 LOCal M%,J,I,DL%,DH%
1520 M%=128
1530 FOR J=1 TO 2
1540  DL%=0:DH%=0
1550  FOR I=1 TO 4
1560   IF (R% && M%)=M% THEN
1570    DL%=FH%+DL%*4
1580    DH%=FL%+DH%*4
1590   ELSE
1600    DL%=BH%+DL%*4
1610    DH%=BL%+DH%*4
1620   END IF
1630   M%=M% DIV 2
1640  END FOR I
1650  POKE A,DL%
1660  POKE A+1,DH%
1670  A=A+2
1680 END FOR J
1699 END DEFine charline

1700 DEFine PROCedure graph(X%,Y%,S%)
1710 CALL do_graph,X%,Y%,S%,C_TAB
1799 END DEFine graph

2000 DEFine PROCedure draw_scene
2010 CLS
2020 draw_snake
2030 draw_pool
2040 IF FOOD=0 THEN FOOD=NUM%
2050 draw_food FOOD
2060 AT 24,3
2064 PRINT "Snake=";SN;
2068 PRINT " Screen=";SC;
2070 PRINT " Score=";STOTAL%+C%
2080 LINE 0,10 TO 256,10
2099 END DEFine draw_scene

2200 DEFine PROCedure draw_pool
2205 LOCal T
2206 T=1
2210 FOR Y=12 TO 14
2220  FOR X=14-T TO 14+T
2230   graph X,Y,12
2240   graph X,Y+(8-(T*2)),12
2250  END FOR X
2255  T=T+1
2260 END FOR Y
```

```
2270 FOR X=11 TO 17
2280  graph X,15,12
2290 END FOR X
2299 END DEFine draw_pool

2500 DEFine PROCedure draw_food (AMOUNT)
2510 IF AMOUNT=0 THEN END DEFine draw_food
2520 FOR K=1 TO AMOUNT
2525  REPeat pos
2530   U%=RND(1 TO 7)+10
2540   V%=RND(1 TO 7)+9
2550   IF pix(V%,U%,3,3)=5 THEN EXIT pos
2560  END REPeat pos
2570  graph U%,V%,6
2575  FX%(K)=U%:FY%(K)=V%
2580 END FOR K
2582 IF AMOUNT<NUM% THEN
2584  FOR K=AMOUNT+1 TO NUM%
2586   FX%(K)=-1
2588  END FOR K
2590 END IF
2599 END DEFine draw_food

2600 DEFine PROCedure draw_snake
2610 A%=2
2620 B%=10
2630 XH%=A%
2640 YH%=B%
2650 H%=0
2660 S%=1
2670 T%=2
2680 graph A%,B%,T%
2690 X%(1)=A%
2700 Y%(1)=B%
2710 FOR Q=2 TO 10
2730  A%=A%+1
2740  graph A%,B%,S%
2750  X%(Q)=A%
2760  Y%(Q)=B%
2770 END FOR Q
2780 A%=A%+1
2790 graph A%,B%,H%
2800 X%(11)=A%
2810 Y%(11)=B%
2820 XV%=1
2830 YV%=0
2840 Z%=1
2850 Q%=11
2899 END DEFine draw_snake
```

```
3000 DEFine PROCedure one_move
3005 IF RND>.5 THEN move_f
3010 get_direction
3020 update
3099 END DEFine one_move

3500 DEFine PROCedure move_f
3505 IF RND<.5 THEN END DEFine move_f
3510 K%=RND(1 TO NUM%)
3520 IF FX%(K%)=-1 THEN  END DEFine move_f
3530 N%=pix(FY%(K%),FX%(K%),4,4)
3540 IF N%=5 THEN
3550  graph FX%(K%),FY%(K%),12
3560 ELSE
3570  IF N%<>0 THEN graph FX%(K%),FY%(K%),11
3580 END IF
3585 REPeat jump
3590  FX%(K%)=FX%(K%)-sgn(13.5-FX%(K%))
3600  FY%(K%)=FY%(K%)-sgn(14.5-FY%(K%))
3610  IF FX%(K%)<0 OR FX%(K%)>31 THEN frog_gone:
      END DEFine move_f
3620  IF FY%(K%)<0 OR FY%(K%)>24 THEN frog_gone:
      END DEFine move_f
3630  N%=pix(FY%(K%),FX%(K%),4,4)
3640  IF N%=6 THEN EXIT jump
3650 END REPeat jump
3660 graph FX%(K%),FY%(K%),5
3699 END DEFine move_f

3700 DEFine FuNction sgn(Q)
3710 IF Q=0 THEN RETurn 0
3720 IF Q>0 THEN
3730  RETurn 1
3740 ELSE
3750  RETurn -1
3760 END IF
3799 END DEFine sgn

3900 DEFine PROCedure frog_gone
3910 FX%(K%)=-1
3920 FOOD=FOOD-1
3930 IF FOOD<>0 THEN END DEFine frog_gone
3940 STOTAL%=STOTAL%+C%
3950 C%=0
3960 IF SC=4 THEN
3962  NO_FOOD=TRUE
3964 ELSE
3970  SC=SC+1
3980  draw_scene
3990 END IF
```

```
3999 END DEFine frog_gone

4000 DEFine PROCedure get_direction
4010 A=KEYROW(1)
4020 IF A=0 THEN END DEFine get_direction
4030 SELect ON A
4040  ON A=4
4050   XV%=0
4060   YV%=-1
4070   H%=7
4080   S%=8
4090  ON A=128
4100   XV%=0
4110   YV%=1
4120   H%=9
4130   S%=10
4140  ON A=2
4150   XV%=-1
4160   YV%=0
4170   H%=3
4180   S%=4
4190  ON A=16
4200   XV%=1
4210   YV%=0
4220   H%=0
4230   S%=1
4240 END SELect
4299 END DEFine get_direction

4500 DEFine PROCedure update
4510 XH%=A%:YH%=B%
4520 A%=A%+XV%:B%=B%+YV%
4530 IF A%>30 THEN A%=1
4540 IF A%<1 THEN A%=30
4550 IF B%>25 THEN B%=0
4560 IF B%<0 THEN B%=25
4570 IF pix(B%,A%,4,4)<>6 THEN hit_f
4580 graph XH%,YH%,S%
4590 graph A%,B%,H%
4600 Q%=Q%+1
4610 IF Q%>MAX% THEN Q%=1
4620 X%(Q%)=A%:Y%(Q%)=B%
4630 graph X%(Z%),Y%(Z%),11
4640 Z%=Z%+1
4650 IF Z%>MAX% THEN Z%=1
4660 graph X%(Z%),Y%(Z%),T%
4699 END DEFine update

5000 DEFine PROCedure hit_f
5010 COL=pix(B%,A%,4,4)
```

```
5020 SELect ON COL
5030  ON COL=4
5040   frog
5050  ON COL=REMAINDER
5060   toad_self
5070  END SELect
5099 END DEFine hit_f

5500 DEFine PROCedure toad_self
5510 BEEP .1,-10
5512 PAUSE 2
5515 BEEP
5520 BEEP .1,-20
5524 PAUSE 2
5526 BEEP
5530 SN=SN+1
5540 IF SN<4 THEN
5560  draw_scene
5570 ELSE
5580  STOTAL%=STOTAL%+C%
5590  NO_LIVES=TRUE
5600 END IF
5699 END DEFine toad_self

5700 DEFine PROCedure frog
5710 BEEP .1,10
5715 PAUSE 2
5720 BEEP
5730 C%=C%+1
5740 FOOD=FOOD-1
5745 rem_f
5750 AT 24,3
5752 PRINT "Snake=";SN;
5754 PRINT " Screen=";SC;
5760 PRINT " Score=";STOTAL%+C%
5770 Z%=Z%-1
5780 IF Z%<1 THEN Z%=MAX%
5790 IF FOOD<>0 THEN END DEFine frog
5800 STOTAL%=STOTAL%+C%
5810 K%=0
5820 IF SC<4 THEN
5830  SC=SC+1
5840  draw_scene
5850 ELSE
5860  NO_FOOD=TRUE
5870 END IF
5899 END DEFine frog

5900 DEFine PROCedure rem_f
5910 K%=0
```

```
5920 REPeat find_f
5930  K%=K%+1
5940  IF A%=FX%(K%) AND B%=FY%(K%) THEN EXIT find_f
5960 END REPeat find_f
5970 FX%(K%)=-1
5999 END DEFine rem_f

6000 DEFine PROCedure super
6010 RESTORE 6010
6020 DATA 8,1,1,0,11,2,11,11,7,11,8,1,1,1
6030 DATA 11,8,1,1,0,11,8,1,1,1
6040 DATA 8,11,11,11,11,10,11,11,8,11,8,11,11,10
6050 DATA 11,8,11,11,11,11,8,11,11,10
6060 DATA 4,4,4,8,11,10,11,11,8,11,8,3,4,10
6070 DATA 11,8,4,8,11,11,8,4,4,10
6080 DATA 11,11,11,8,11,10,11,11,8,11,8,11,11,11
6090 DATA 11,8,1,1,11,11,8,10,1,11
6092 DATA 11,11,11,8,11,10,11,11,8,11,8,11,11,11
6094 DATA 11,8,11,11,11,11,8,11,10,1
6100 DATA 2,1,1,1,11,10,1,1,1,11,2,11,11,11,11
6110 DATA 4,4,4,2,11,2,11,11,9
6120 PAUSE 20
6130 CLS
6140 FOR I=2 TO 7
6150  FOR J=3 TO 26
6160   READ S%
6170   graph J,I,S%
6180  END FOR J
6190 END FOR I
6199 END DEFine super

7000 DEFine PROCedure end_game
7010 CLS
7015 AT 3,10
7016 STOTAL=STOTAL%
7020 SELect ON STOTAL
7030  ON STOTAL=0 TO 9
7040   PRINT "Demoted to slowworm"
7050  ON STOTAL=10 TO 19
7060   PRINT "The frogs don't have to worry"
7070  ON STOTAL=20 TO 29
7080   PRINT "Not bad for a grass snake"
7090  ON STOTAL=30 TO 34
7100   PRINT "Well slithered"
7110  ON STOTAL=REMAINDER
7140   IF NO_FOOD THEN
7150    super
7160    snake
7170   ELSE
7180    PRINT "A venomous performance"
```

```
7190   END IF
7200 END SELect
7210 FOR I= 5 TO 1 STEP -1
7220  IF STOTAL%>=T(I) THEN N=I
7230 END FOR I
7235 AT 15,10
7240 IF N=0 THEN
7250  PRINT "You scored ";STOTAL%
7260 ELSE
7270  PRINT "You are now ranked ";N
7280  PRINT TO 10;"in the snake league"
7290 IF N<5 THEN
7300  FOR I=5 TO N+1 STEP -1
7310   T(I)=T(I-1)
7320   T$(I)=T$(I-1)
7330  END FOR I
7340 END IF
7350 T(N)=STOTAL%
7355 PRINT TO 10;
7360 INPUT "What is your name ";T$(N)
7370 CLS
7380 FOR I=1 TO 5
7390  AT I*2+5,5
7400  PRINT T$(I,1 TO 15);
7410  PRINT TO 21; T(I)
7420 END FOR I
7440 REPeat an_game
7445  AT 21,10
7450  INPUT "Another slither ";A$
7455  IF A$="Y" OR A$="N" THEN EXIT an_game
7460  END REPeat an_game
7470 IF A$="N" THEN
7480  AGAIN=FALSE
7490 ELSE
7500  AGAIN=TRUE
7510 END IF
7599 END DEFine end_game

8000 DEFine PROCedure title
8010 PAPER 6
8020 CLS
8030 INK 0
8040 snake
8045 CLS
8050 AT 4,2
8060 PRINT "You must guide your snake using"
8065 PRINT "  the four arrow keys and"
8070 PRINT "  eat all the green frogs";
8080 graph 19,8,5
8085 PRINT \\
```

```
8090 PRINT " Don't fall in the water or try to"
8110 PRINT "  eat yourself or you will lose"
8130 PRINT "  one of your five lives"\\
8140 PRINT "  You get five screens"
8145 PRINT "  of frogs to eat before you"
8150 PRINT "  become a SUPER SNAKE"
8155 REPeat wait
8156  AT 23,7
8160  FLASH 1
8170  PRINT "PRESS ANY KEY"
8180  FLASH 0
8190  IF INKEY$<>"" THEN EXIT wait
8195 END REPeat wait
8199 END DEFine title

8500 DEFine PROCedure snake
8510 DATA 8,1,1,0,11,8,1,1,1,11,8,1,1,1,11
8520 DATA 2,11,8,0,11,8,1,1,0
8530 DATA 8,11,11,11,11,8,11,11,10,11,8,11,11,10
8540 DATA 11,10,8,1,11,11,8,11,11,11
8545 DATA 4,4,4,8,11,8,11,11,10,11,8,1,1
8550 DATA 10,11,10,8,11,11,11,4,4,8,11
8560 DATA 11,11,11,8,11,8,11,11,10,11,8,10,1,1
8570 DATA 11,10,4,8,11,11,8,1,1,11
8580 DATA 11,11,11,8,11,8,11,11,10,11,8,11,11
8590 DATA 10,11,10,11,4,8,11,8,11,11,11
8600 DATA 2,1,1,1,11,2,11,11,9,11,2,11,11,9
8610 DATA 11,9,11,11,2,11,4,4,4,2
8620 RESTORE 8500
8630 FOR I=9 TO 14
8640  FOR J=3 TO 26
8650   READ S
8660   graph J,I,S
8670  END FOR J
8680 END FOR I
8690 PAUSE 100
8699 END DEFine snake

9500 DEFine FuNction pix(R%,C%,X%,Y%)
9510 LOCal A%,P%,B%,A
9520 P%=C%*8+X%
9530 A=131072+(512*R%+Y%*64+P% DIV 4)*2
9540 B%=PEEK(A+1)
9550 A%=PEEK(A)
9560 P%=P% MOD 4
9570 IF P%=3 THEN RETurn ((2 && A%)*2)+(3 && B%)
9580 IF P%=2 THEN RETurn ((8 && A%)DIV 2)+
     ((12 && B%) DIV 4)
9590 IF P%=1 THEN RETurn ((32 && A%)DIV 8)+
     ((48 && B%) DIV 16)
```

```
9600 IF P%=0 THEN RETurn ((128 && A%) DIV 32)+
     ((192 && B%) DIV 64)
9699 END DEFine pix
```

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, but they also 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 is in the construction of the graphics and the board; however, many of the techniques of animation introduced in earlier chapters will again 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 the 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 forwards 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.

The only real problem in constructing the board is making sure that the 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

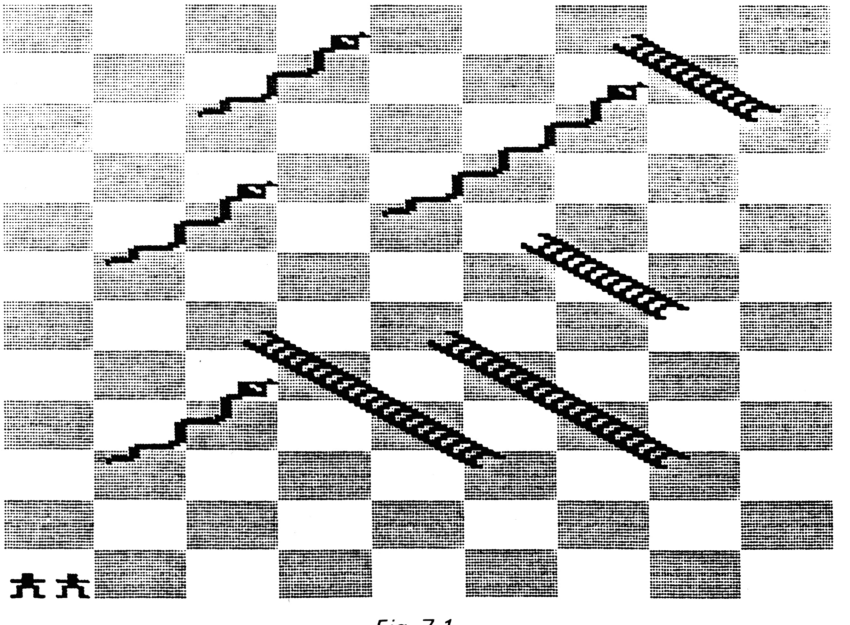


Fig. 7.1.

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 character 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 the screen. However, although this method is simple in theory, finding out what character is present at any position on the QL's screen is rather difficult and so for Snakes and Ladders the array storage method is used. In other words, every character that is printed to the screen to make up the board is also stored in an array which is later used to find out what character is at any given board position.

Main program

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

```
 10 REMark Snakes and Ladders
 15 PAPER 1
 20 WINDOW 512,256,0,0
 30 init_graph
 40 REPeat game
 50  title
 60  p_sandl
 70  start
 80  I%=0
 90  REPeat thrw
100   throw I%
110   IF HOME THEN EXIT thrw
120   I%=NOT(I%)
130  END REPeat thrw
140  end_game
150  IF AGAIN=FALSE THEN EXIT game
160 END REPeat game
199 STOP
```

The first two procedures, *init_graph* and *title* should be familiar if you have read earlier chapters, but notice that in this case *title* has to be used after *init_graph* because it uses some of the graphics characters that *init_graph* defines. After *init_graph* and *title* have been called, line 60 calls *p_sandl* which prints the Snakes and Ladders board on the screen. Following this, *start* is called to

initialise some of the variables before the game begins. The REPeat loop, lines 90 to 130, is responsible for running the game. Each time through the loop both players make one move. Line 100 calls *throw* to allow player I% to make a 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 140 calls *end_game* and line 150 either restarts the entire game or brings it to an end, depending on the state of AGAIN.

Procedures *init_graph* and *title*

Procedure *init_graph* simply sets up the graphics characters and arrays used in Snakes and Ladders:

```
350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
430 READ B
440  POKE_W I,B
450 END FOR I
460 C_TAB=RESPR(32*23)
465 FOR Q=0 TO 1
470  def_graph 11*Q+0,6,1+Q,0,2,127,102,16,124,252,
     192
480  def_graph 11*Q+1,6,1+Q,3,3,3,3,3,7,254,252
490  def_graph 11*Q+2,6,1+Q,0,0,0,0,0,0,1,3
500  def_graph 11*Q+3,6,1+Q,3,3,31,62,32,0,0,0
510  def_graph 11*Q+4,6,1+Q,0,0,6,3,1,35,54,28
520  def_graph 11*Q+5,6,1+Q,152,60,102,207,217,115,
     54,28
530  def_graph 11*Q+6,6,1+Q,0,0,0,0,128,192,96,240
540  def_graph 11*Q+7,6,1+Q,13,7,3,1,0,0,0,0
550  def_graph 11*Q+8,6,1+Q,152,60,102,192,192,96,
     0,0
560  def_graph 11*Q+9,3,1+Q,24,24,126,24,60,36,36,
     102
570  def_graph 11*Q+10,1+Q,1+Q,255,255,255,255,255,
     255,255,255
580 END FOR Q
590 def_graph 22,6,1,0,0,0,24,24,0,0,0
600 TRUE=1
```

```
610 FALSE=0
620 DIM board%(20,26)
630 DIM VX%(1)
640 DIM N$(1,20)
650 DIM X%(1)
660 DIM Y%(1)
699 END DEFine init_graph
```

The way the graphics characters go together to make a snake or a ladder is quite complicated and can best be understood from Fig. 7.2. The game demands a total of twenty-three combinations of shapes and foreground/background colours, and Table 7.1 summarises them.

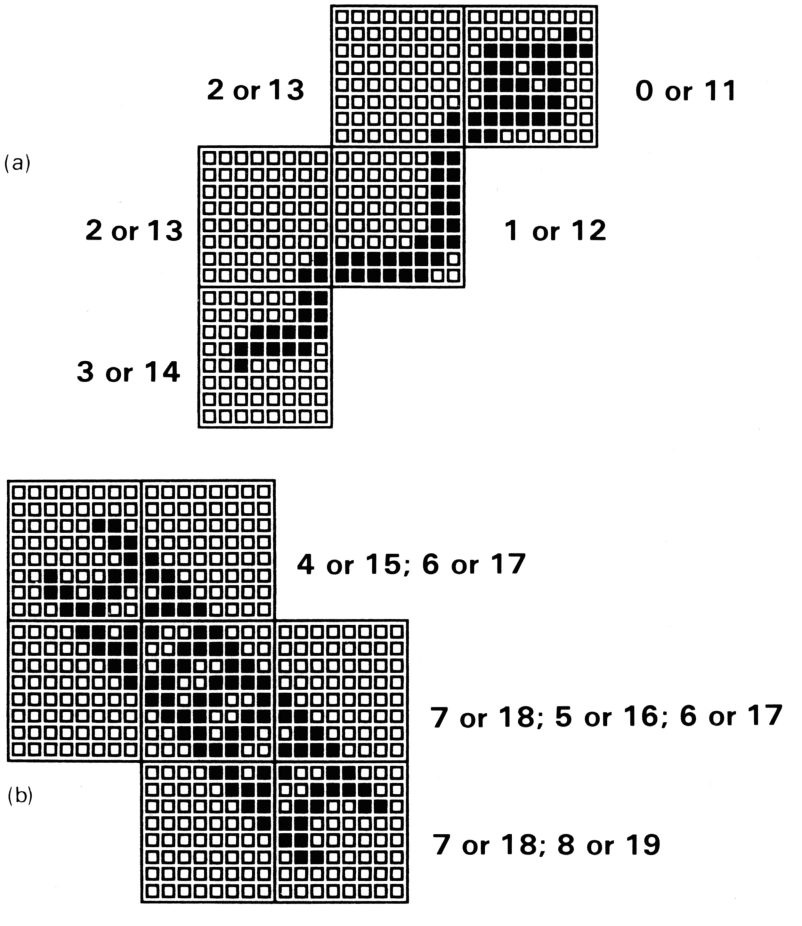


Fig. 7.2. Graphics characters for (a) snake and (b) ladder.

Table 7.1. Graphics characters

Number	Shape	Foreground	Background
0	snake head	yellow	blue
1	snake body	yellow	blue
2	corner	yellow	blue
3	snake tail	yellow	blue
4	ladder top	yellow	blue
5	ladder middle	yellow	blue
6	ladder right	yellow	blue
7	ladder left	yellow	blue
8	ladder bottom	yellow	blue
9	man	magenta	blue
10	space	blue	blue
11	snake head	yellow	red
12	snake body	yellow	red
13	corner	yellow	red
14	snake tail	yellow	red
15	ladder top	yellow	red
16	ladder middle	yellow	red
17	ladder right	yellow	red
18	ladder left	yellow	red
19	ladder bottom	yellow	red
20	man	magenta	red
21	space	red	red
22	dot for dice	yellow	blue

Notice how each shape has to be made available in two background colours so that it can be printed on either type of board square. In general, if S is the number of a graphics character with a blue background, then S+11 is the same shape on a red background. The arrays X% and Y% are used to hold the co-ordinates of the man shape (graphics character 9) for each player. What the array VX% is 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.

Procedure *title* 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.

```
8000 DEFine PROCedure title
8005 CLS
8006 RANDOMISE
8010 AT 4,14:PRINT "and"
8020 sl
8030 AT 12,3
8040 PRINT "This is a game for two players"
8050 PRINT "  The first player to reach the top"
8060 PRINT "  left of the board wins!!"
8070 AT 21,2
8080 REPeat real
8090  INPUT "Do you want to use a real dice? ";A$
8100  IF A$="Y" OR A$="N" THEN EXIT real
8110 END REPeat real
8120 IF A$="Y" THEN
8130  DICE=TRUE
8140 ELSE
8150  DICE=FALSE
8160 END IF
8170 AT 22,3
8180 INPUT "What is the first players name ?";N$(0)
8190 INPUT "   What is the second players name ?";
     N$(1)
8199 END DEFine title
```

Lines 8080 to 8160 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 8180 and 8190 get the players' names and store them into N$(0) and NS(1). Line 8020 calls procedure *sl* which uses some of the graphics characters defined by *init_graph* to generate a title line involving snakes and ladders:

```
8200 DEFine PROCedure sl
8210 X1%=1
8220 X2%=16
8230 FOR Y=5 TO 2 STEP -1
8240  graph X1%,Y,2
8250  graph X1%+1,Y,1
8260  graph X1%+3,Y,2
8270  graph X1%+4,Y,1
8280  graph X2%,Y,7
8290  graph X2%+1,Y,
8300  graph X2%+2,Y,6
8310  graph X2%+4,Y,7
8320  graph X2%+5,Y,5
8330  graph X2%+6,Y,6
8340  X2%=X2%-1
8350  X1%=X1%+1
```

```
8355 END FOR Y
8360 graph 5,1,2
8370 graph 6,1,0
8380 graph 8,1,2
8390 graph 9,1,0
8400 graph 13,1,4
8410 graph 14,1,6
8420 graph 17,1,4
8430 graph 18,1,6
8440 graph 1,6,3
8450 graph 4,6,3
8460 graph 17,6,7
8470 graph 18,6,8
8480 graph 21,6,7
8490 graph 22,6,8
8499 END DEFine s1
```

Printing the board – *p_sandl* and its associated procedures

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

```
2000 DEFine PROCedure p_sandl
2010 p_squares
2020 FOR S=1 TO RND(4 TO 6)
2030  make_snake
2040  make_ladder
2050 END FOR S
2099 END DEFine p_sandl
```

Procedure *p_squares* prints the pattern of blue and red squares that makes up the board. Procedures *make_snake* and *make_ladder* print a single snake and a single ladder respectively. The FOR loop, lines 2020 to 2050, calls *make_snake* and *make_ladder* a random number of times between 4 and 6.

Procedure *p_squares* is fairly straightforward:

```
2500 DEFine PROCedure p_squares
2505 PAPER 6
2510 CLS
2530 FOR Y=2 TO 24 STEP 2
2540  FOR X=2 TO 19 STEP 2
2560   set_graph X,Y,10
2570   set_graph X+1,Y,10
2580   set_graph X,Y+1,10
2590   set_graph X+1,Y+1,10
2600  END FOR X
```

```
2610 END FOR Y
2699 END DEFine p_squares
```

Each of the squares that make up the board is composed of four blanks. These are alternately printed in blue and red by using the procedure *set_graph*:

```
1800 DEFine PROCedure set_graph(X%,Y%,S%)
1810 IF ( (X%+1)/2 MOD 2)=( (Y%+1)/2 MOD 2) THEN
     S%=S%+11
1820 graph X%,Y%,S%
1830 board%(X%,Y%)=S%
1899 END DEFine set_graph
```

This procedure simply prints blue or red blocks according to whether the position on the board corresponds to an odd or even row or column. Procedure *set_graph* also stores in the array board the code of the character printed on the screen. That is, if all the characters are printed using *set_graph* then board(X%,Y%) contains the code of the character currently at X%,Y% on the screen.

Procedure *make_snake* looks a little complicated but this is only because of the large number of calls to *set_graph* necessary to print all the characters that make up a snake.

```
3000 DEFine PROCedure make_snake
3005 LOCal A%,B%
3010 TRY=0
3015 REPeat poss
3020  A%=RND(3 TO 8):B%=RND(1 TO 8)
3030  IF A%-1<(10-B%) THEN
3040   L%=RND(2 TO A%-1)+1
3050  ELSE
3060   L%=RND(2 TO 10-B%)+1
3070  END IF
3080  IF TRY>20 THEN
3090   END DEFine make_snake
3100  ELSE
3110   TRY=TRY+1
3120  END IF
3130  IF clash(A%,B%,L%,-1)=FALSE THEN EXIT poss
3140 END REPeat poss
3150 set_graph A%*2+1,B%*2+1,0
3160 set_graph A%*2,B%*2+1,2
3170 FOR Z=A%-1 TO A%-L%+2 STEP -1
3180  B%=B%+1
3190  set_graph Z*2+2,B%*2,1
3200  set_graph Z*2+1,B%*2,2
3210  set_graph Z*2+1,B%*2+1,1
```

```
3220  set_graph Z*2,B%*2+1,2
3230 END FOR Z
3240 B%=B%+1
3250 set_graph (A%-L%+1)*2+2,B%*2,3
3299 END DEFine make_snake
```

Line 3020 generates two random numbers that specify the starting position of a snake. Lines 3030 to 3070 then generate 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 3130 uses the function *clash* to check whether or not the snake specified by A%, B% and L% crosses any other snake or a ladder. If it does, then another set of random numbers is generated and the process repeated. In this way various starting positions and lengths of snake are tried until one is found that 'fits' – that is, 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, so the variable TRY is used to count the number of attempts. Lines 3080 to 3120 make *make_snake* give up after 20 tries. Once a snake that fits has been found then lines 3150 to 3250 print the combination of characters that make it appear on the board. Procedure *set_graph* is used rather than *graph* 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 *set_graph* will use the graphics character with a red background, and where it is positioned in a blue square *set_graph* will use the graphics character with a blue background.

Procedure *make_ladder* works in roughly the same way as *make_snake*:

```
3500 DEFine PROCedure make_ladder
3505 LOCal A%,B%
3510 TRY=0
3515 REPeat poss
3520  A%=RND(1 TO 6)
3530  B%=RND(1 TO 8)
3540  IF TRY>20 THEN
3550   END DEFine make_ladder
3560  ELSE
3570   TRY=TRY+1
3580  END IF
3590  IF 9-A%<11-B% THEN
3600   L%=RND (2 TO 9-A%)+1
3610  ELSE
3620   L%=RND(2 TO 11-B%)+1
```

```
3630  END IF
3640  IF clash(A%,B%,L%,1)=FALSE THEN EXIT poss
3650 END REPeat poss
3660 set_graph A%*2+1,B%*2+1,4
3670 set_graph A%*2+2,B%*2+1,6
3680 set_graph A%*2+1,B%*2+2,7
3690 B%=B%+1
3700 FOR Z=A%+1 TO A%+L%-2
3710  set_graph Z*2,B%*2,5
3720  set_graph Z*2+1,B%*2+1,5
3730  set_graph Z*2+2,B%*2+1,6
3740  set_graph Z*2+1,B%*2+2,7
3750  set_graph Z*2+1,B%*2,6
3760  set_graph Z*2,B%*2+1,7
3770  B%=B%+1
3780 END FOR Z
3790 set_graph (A%+L%-1)*2,B%*2,8
3799 END DEFine make_ladder
```

Lines 3515 to 3650 find values of A%, B% and L% that produce a ladder that doesn't interfere with any ladders or snakes already printed on the screen. Lines 3660 to 3790 print the combination of characters that produces a ladder.

The function *clash* works by examining the array board to find out what is already printed on the screen in every character location that would be affected by printing a ladder or a snake.

```
9000 DEFine FuNction clash(X%,Y%,L%,D%)
9010 LOCal T%,A%,B%
9015 T%=FALSE
9020 E%=2*(X%+D%*L%)+D%
9030 A%=X%*2-D%
9040 B%=Y%*2-1
9050 REPeat sq
9060  FOR Q=-1 TO 1
9070   IF board%(A%+Q+1,B%+1)<>10 AND
       board%(A%+Q+1,B%+1)<>21 THEN T%=TRUE
9090  END FOR Q
9100  A%=A%+D%
9110  B%=B%+1
9120  IF A%=E% OR T%=TRUE THEN EXIT sq
9130 END REPeat sq
9140 RETurn T%
9199 END DEFine clash
```

Lines 9020 to 9040 work out which character locations have to be examined. Lines 9050 to 9130 then proceed to examine each location in turn until either the last location has been examined or until a non-

blank character is found. Line 9070 is responsible for the actual check that the screen location contains a blank. Notice that as there are two blank graphics characters, i.e. a blue and a red block, both have to be tested for.

Procedure *start*

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

```
3800 DEFine PROCedure start
3810 FOR I=0 TO 1
3820  X%(I)=1:Y%(I)=12
3840  VX%(I)=1
3850  graph X%(I)*2-I+1,Y%(I)*2+1,9
3860 END FOR I
3870 HOME=FALSE
3899 END DEFine start
```

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+1,Y%(I%)*2+1 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%)
```

In all the earlier games, any character that moved did so against a clear background and so each time through the animation loop it could be blanked out using a space character of the appropriate colour. However, the man characters in Snakes and Ladders move on a background that consists of a wide range of different characters. This complicates the move logic a little but as will be explained in the next section, the use of the array 'board' produces a very simple method of allowing the man to move without destroying the board display.

Moving men - *throw* and its associated procedures

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

```
5000 DEFine PROCedure throw(I%)
5010 IF DICE THEN
5020  dice1
5030 ELSE
5040  dice2
5050 END IF
5060 REPeat mve
5070  one_move I%
5080  BEEP .1,-10*I%
5090  PAUSE 2
5100  BEEP
5110  M%=M%-1
5120  IF M%=0 THEN EXIT mve
5130  HOME=(X%(I%)=1 AND Y%(I%)=1)
5140  IF HOME THEN EXIT mve
5150 END REPeat mve
5160 IF SNAKE AND LADDER THEN choose
5170 IF SNAKE THEN updown I%,1,3
5180 IF LADDER THEN updown I%,-1,4
5190 HOME=(X%(I%)=1 AND Y%(I%)=1)
5199 END DEFine throw
```

Depending on the value of DICE, lines 5010 to 5050 call one of the two dice procedures, either to ask player I% the result of throwing a real dice (*dice1*) or to generate a random number to simulate the throw of a dice (*dice2*). No matter which dice procedure is called, the result is returned in M% and lines 5060 to 5150 then move man I% by M% squares.

A move of a single square is produced by calling *one_move* (line 5070) and line 5130 checks to see if the move has resulted in the man reaching the top left-hand square. After each call, *one_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 5160 to 5180 check for the man landing on a snake or a ladder as a result of the last move. Line 5160 resolves the problem caused by landing on a square that contains both a snake's head and the foot of a ladder by calling *choose*, which chooses *either* the ladder *or* the snake at random. Procedure *updown* deals with making the man either slide down a snake or climb up a ladder (lines 5170 and 5180).

Procedure *one_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 or the start of a ladder within the new square.

```
4000 DEFine PROCedure one_move(I%)
4010 graph X%(I%)*2-I%+1,Y%(I%)*2+1,
     board%(X%(I%)*2-I%+1,Y%(I%)*2+1)
4020 X%(I%)=X%(I%)+VX%(I%)
4030 IF X%(I%)>9 THEN
4040  X%(I%)=9
4050  Y%(I%)=Y%(I%)-1
4060  VX%(I%)=-VX%(I%)
4070 END IF
4080 IF X%(I%)<1 THEN
4090  X%(I%)=1
4100  Y%(I%)=Y%(I%)-1
4110  VX%(I%)=-VX%(I%)
4120 END IF
4130 IF board%(X%(I%)*2+1,Y%(I%)*2+1)=0 OR
     board%(X%(I%)*2+1,Y%(I%)*2+1)=11 THEN
4140  SNAKE=TRUE
4150 ELSE
4160  SNAKE=FALSE
4170 END IF
4180 IF board%(X%(I%)*2,Y%(I%)*2)=8 OR
     board%(X%(I%)*2,Y%(I%)*2)=19 THEN
4190  LADDER=TRUE
4200 ELSE
4210  LADDER=FALSE
4220 END IF
4230 p_man X%(I%)*2-I%+1,Y%(I%)*2+1
4299 END DEFine one_move
```

The blanking of the man character at its old position is taken care of by line 4010, which prints the character stored in the array 'board' back at its correct position on the screen. The update of the co-ordinates of the man character is implemented by lines 4030 to 4120. You should be able to recognise these lines as implementing 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 his direction of motion is reversed (i.e. VX%(I%)=−VX%(I%)). Finally, lines 4130 to 4220 check for a snake's head and the foot of a ladder respectively and then line 4230 prints the man at his new position using *p_man*.

The procedure *p_man* is very similar to *set_graph* in that it selects the man character with the correct background colour for the board

square that it is about to be printed on, but it doesn't store the character code in the array 'board%'. The reason for this difference should be obvious in that the purpose of 'board%' is to store the character that was present at the location before the man character was printed over it!

```
4500 DEFine PROCedure p_man(X%,Y%)
4505 LOCal S%
4506 S%=9
4510 IF ((X%+1)/2 MOD 2)=((Y%+1)/2 MOD 2) THEN
     S%=S%+11
4520 graph X%,Y%,S%
4599 END DEFine p_man
```

This leaves a number of small procedures and one large procedure, *updown*, to be described. Before moving on to *updown* it is better to deal with the smaller procedures.

Procedure *dice1* asks for the result of throwing a real dice:

```
5500 DEFine PROCedure dice1
5510 AT 21,2
5520 PRINT "It's your throw ";N$(I%);"     "
5530 PRINT "-";N$(I%)
5550 REPeat DO_DICE
5555  AT 22,2
5560  INPUT "What did you get ";M%
5570  IF M%>0 AND M%<7 THEN EXIT DO_DICE
5580 END REPeat DO_DICE
5599 END DEFine dice1
```

Procedure *dice2* uses the RND function to supply a random number instead of throwing a real dice. Instead of simply printing the result, however, it also produces an animated dice with the dots changing randomly. This is an attempt to recreate the suspense of throwing and waiting to see the result of a real dice. Procedure *dice2* also calls a number of other procedures to print the various dot patterns:

```
5600 DEFine PROCedure dice2
5610 ROLL%=RND(5 TO 15)
5620 FOR Q=25 TO 27
5630  FOR R=25 TO 27
5640   graph Q,R,10
5650  END FOR R
5660 END FOR Q
5670 graph 26,26,22
5680 M=1
5690 AT 21,3:PRINT "It's your throw"
5700 PRINT "  -";N$(I%);"         "
```

```
5710 PRINT "  PRESS ANY KEY"
5720 PAUSE
5725 graph 26,26,10
5730 FOR Q=1 TO ROLL%
5735 FOR C=22,10
5736 IF C=10 THEN PAUSE Q*2
5737 IF C=10 AND Q=ROLL% THEN EXIT Q
5745  C%=C
5750  SELect ON M
5760   ON M=1
5770    one C%
5780   ON M=2
5790    two C%
5800   ON M=3
5810    one C%
5820    two C%
5840   ON M=4
5850    two C%
5860    four C%
5870   ON M=5
5880    one C%
5890    two C%
5900    four C%
5910   ON M=6
5920    two C%
5930    four C%
5940    six C%
5950  END SELect
5955 END FOR C
5960  M=RND(1 TO 6)
5980 END FOR Q
5990 M%=M
5999 END DEFine dice2

6000 DEFine PROCedure one(C%)
6010 graph 26,26,C%
6099 END DEFine one

6100 DEFine PROCedure two(C%)
6110 graph 25,25,C%
6120 graph 27,27,C%
6199 END DEFine two

6200 DEFine PROCedure four(C%)
6210 graph 25,27,C%
6220 graph 27,25,C%
6299 END DEFine four

6300 DEFine PROCedure six(C%)
6310 graph 25,26,C%
```

```
6320 graph 27,26,C%
6399 END DEFine six
```

Note the use of the rather unusual FOR statement in line 5735 which repeats the dice printing twice, once using the dot character (code 22) and once using a blue block (code 10) to blank out the dot pattern.

Procedure *choose* sets one of SNAKE or LADDER to FALSE at random, so choosing one of the two possibilities:

```
5200 DEFine PROCedure choose
5210 IF RND>.5 THEN
5220  LADDER=FALSE
5230 ELSE
5240  SNAKE=FALSE
5299 END DEFine choose
```

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

```
6500 DEFine PROCedure updown(I%,D%,C%)
6510 LOCal A%,B%
6520 graph X%(I%)*2-I%+1,Y%(I%)*2+1,
     board%(X%(I%)*2-I%+1,Y%(I%)*2+1)
6530 A%=X%(I%)*2+1
6540 B%=Y%(I%)*2+1
6550 REPeat slide
6560  p_man A%,B%
6570  BEEP .1,2*B%
6580  PAUSE 2
6590  BEEP
6600  graph A%,B%,board%(A%,B%)
6605 A%=A%-1:B%=B%+D%
6610  IF board%(A%,B%)=C% OR
      board%(A%,B%)=C%+11 THEN EXIT slide
6620 END REPeat slide
6630 A%=A%+1:B%=B%-D%
6640 graph A%,B%,board%(A%,B%)
6650 X%(I%)=A% DIV 2-1
6660 Y%(I%)=B% DIV 2-1
6670 IF D%=1 THEN
6680  Y%(I%)=Y%(I%)+2
6690  X%(I%)=X%(I%)+1
6700 END IF
6705 FOR I=0 TO 1
6710  p_man X%(I)*2-I+1,Y%(I)*2+1
6715 END FOR I
6720 IF (Y%(I%) DIV 2)*2=Y%(I%) THEN
6730  VX%(I%)=1
6740 ELSE
```

```
6750  VX%(I%)=-1
6760 END IF
6799 END DEFine updown
```

The way *updown* works is simply by printing the man character at each position on the snake or the ladder using the array 'board' to blank out the old version of the man (lines 6520, 6600 and 6640), making appropriate sound effects on the way until the character that marks the end of the snake or ladder is found (line 6610). The value of D% controls whether the man is moved up a ladder (D%=1) or down a snake (D%=−1) and C% is the code of the final character of a ladder or a snake.

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 a snake's head and the foot of a ladder occur within the same square was not anticipated! 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, it should be easy to add new procedures to cope with situations as they arise.

Procedure *end_game*

Procedure *end_game* simply congratulates the player who is first home and asks if another game is required:

```
8500 DEFine PROCedure end_game
8510 AT 21,2
8520 PRINT "You win ";N$(I%);"          "
8530 REPeat an_game
8540  INPUT "  Another game Y/N ?";A$
8550  IF A$="Y" OR A$="N" THEN EXIT an_game
8560 END REPeat an_game
8570 IF A$="Y" THEN
8580  AGAIN=TRUE
8590 ELSE
8600  AGAIN=FALSE
8610 END IF
8699 END DEFine end_game
```

Evaluation

The final program works reasonably quickly and for a 32 by 32 character screen it 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. The biggest problem is that the outcome of the game is entirely a matter of chance and although 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 REMark Snakes and Ladders
 15 PAPER 1
 20 WINDOW 512,256,0,0
 30 init_graph
 40 REPeat game
 50  title
 60  p_sandl
 70  start
 80  I%=0
 90  REPeat thrw
100   throw I%
110   IF HOME THEN EXIT thrw
120   I%=NOT(I%)
130  END REPeat thrw
140  end_game
150  IF AGAIN=FALSE THEN EXIT game
160 END REPeat game
199 STOP

350 DEFine PROCedure init_graph
360 DATA -5245,-10109,-7806,-11646
370 DATA -6783,1665,2,0,8769,8316
380 DATA 0,0,9264,18432,8834,-11524,128
390 DATA 22664,-20228,32,26350,28672,20085
400 do_graph=RESPR(100)
410 RESTORE 360
420 FOR I=do_graph TO do_graph+22*2  STEP 2
```

```
430 READ B
440  POKE_W I,B
450 END FOR I
460 C_TAB=RESPR(32*23)
465 FOR Q=0 TO 1
470  def_graph 11*Q+0,6,1+Q,0,2,127,102,16,124,
     252,192
480  def_graph 11*Q+1,6,1+Q,3,3,3,3,3,7,254,252
490  def_graph 11*Q+2,6,1+Q,0,0,0,0,0,0,1,3
500  def_graph 11*Q+3,6,1+Q,3,3,31,62,32,0,0,0
510  def_graph 11*Q+4,6,1+Q,0,0,6,3,1,35,54,28
520  def_graph 11*Q+5,6,1+Q,152,60,102,207,217,115,
     54,28
530  def_graph 11*Q+6,6,1+Q,0,0,0,0,128,192,96,240
540  def_graph 11*Q+7,6,1+Q,13,7,3,1,0,0,0,0
550  def_graph 11*Q+8,6,1+Q,152,60,102,192,192,96,
     0,0
560  def_graph 11*Q+9,3,1+Q,24,24,126,24,60,36,36,
     102
570  def_graph 11*Q+10,1+Q,1+Q,255,255,255,255,255,
     255,255,255
580 END FOR Q
590 def_graph 22,6,1,0,0,0,24,24,0,0,0
600 TRUE=1
610 FALSE=0
620 DIM board%(20,26)
630 DIM VX%(1)
640 DIM N$(1,20)
650 DIM X%(1)
660 DIM Y%(1)
699 END DEFine init_graph

1300 DEFine PROCedure def_graph(S%,F%,B%,
     R0%,R1%,R2%,R3%,R4%,R5%,R6%,R7%)
1310 LOCal A,FH%,FL%,BH%,BL%
1320 A=C_TAB+32*S%
1330 FH%=(F% DIV 4)*2
1340 FL%=F% && 3
1350 BH%=(B% DIV 4)*2
1360 BL%=B% && 3
1370 charline(R0%)
1380 charline(R1%)
1390 charline(R2%)
1400 charline(R3%)
1410 charline(R4%)
1420 charline(R5%)
1430 charline(R6%)
1440 charline(R7%)
1499 END DEFine def_graph
```

```
1500 DEFine PROCedure charline(R%)
1502 LOCal M%,J,I,DL%,DH%
1503 M%=128
1504 FOR J=1 TO 2
1505  DL%=0:DH%=0
1510  FOR I=1 TO 4
1520   IF (R% && M%)=M% THEN
1530    DL%=FH%+DL%*4
1540    DH%=FL%+DH%*4
1550   ELSE
1560    DL%=BH%+DL%*4
1570    DH%=BL%+DH%*4
1580   END IF
1590   M%=M% DIV 2
1600  END FOR I
1610  POKE A,DL%
1620  POKE A+1,DH%
1630  A=A+2
1640 END FOR J
1699 END DEFine charline

1700 DEFine PROCedure graph(X%,Y%,S%)
1710 CALL do_graph,X%,Y%,S%,C_TAB
1799 END DEFine graph

1800 DEFine PROCedure set_graph(X%,Y%,S%)
1810 IF ( (X%+1)/2 MOD 2)=( (Y%+1)/2 MOD 2) THEN
     S%=S%+11
1820 graph X%,Y%,S%
1830 board%(X%,Y%)=S%
1899 END DEFine set_graph

2000 DEFine PROCedure p_sand1
2010 p_squares
2020 FOR S=1 TO RND(4 TO 6)
2030  make_snake
2040  make_ladder
2050 END FOR S
2099 END DEFine p_sand1

2500 DEFine PROCedure p_squares
2505 PAPER 6
2510 CLS
2530 FOR Y=2 TO 24 STEP 2
2540  FOR X=2 TO 19 STEP 2
2560   set_graph X,Y,10
2570   set_graph X+1,Y,10
2580   set_graph X,Y+1,10
2590   set_graph X+1,Y+1,10
2600  END FOR X
```

```
2610 END FOR Y
2699 END DEFine p_squares

3000 DEFine PROCedure make_snake
3005 LOCal A%,B%
3010 TRY=0
3015 REPeat poss
3020  A%=RND(3 TO 8):B%=RND(1 TO 8)
3030  IF A%-1<(10-B%) THEN
3040   L%=RND(2 TO A%-1)+1
3050  ELSE
3060   L%=RND(2 TO 10-B%)+1
3070  END IF
3080  IF TRY>20 THEN
3090   END DEFine make_snake
3100  ELSE
3110   TRY=TRY+1
3120  END IF
3130  IF clash(A%,B%,L%,-1)=FALSE THEN EXIT poss
3140 END REPeat poss
3150 set_graph A%*2+1,B%*2+1,0
3160 set_graph A%*2,B%*2+1,2
3170 FOR Z=A%-1 TO A%-L%+2 STEP -1
3180  B%=B%+1
3190  set_graph Z*2+2,B%*2,1
3200  set_graph Z*2+1,B%*2,2
3210  set_graph Z*2+1,B%*2+1,1
3220  set_graph Z*2,B%*2+1,2
3230 END FOR Z
3240 B%=B%+1
3250 set_graph (A%-L%+1)*2+2,B%*2,3
3299 END DEFine make_snake

3500 DEFine PROCedure make_ladder
3505 LOCal A%,B%
3510 TRY=0
3515 REPeat poss
3520  A%=RND(1 TO 6)
3530  B%=RND(1 TO 8)
3540  IF TRY>20 THEN
3550   END DEFine make_ladder
3560  ELSE
3570   TRY=TRY+1
3580  END IF
3590  IF 9-A%<11-B% THEN
3600   L%=RND (2 TO 9-A%)+1
3610  ELSE
3620   L%=RND(2 TO 11-B%)+1
3630  END IF
3640  IF clash(A%,B%,L%,1)=FALSE THEN EXIT poss
```

```
3650 END REPeat poss
3660 set_graph A%*2+1,B%*2+1,4
3670 set_graph A%*2+2,B%*2+1,6
3680 set_graph A%*2+1,B%*2+2,7
3690 B%=B%+1
3700 FOR Z=A%+1 TO A%+L%-2
3710  set_graph Z*2,B%*2,5
3720  set_graph Z*2+1,B%*2+1,5
3730  set_graph Z*2+2,B%*2+1,6
3740  set_graph Z*2+1,B%*2+2,7
3750  set_graph Z*2+1,B%*2,6
3760  set_graph Z*2,B%*2+1,7
3770  B%=B%+1
3780 END FOR Z
3790 set_graph (A%+L%-1)*2,B%*2,8
3799 END DEFine make_ladder

3800 DEFine PROCedure start
3810 FOR I=0 TO 1
3820  X%(I)=1:Y%(I)=12
3840  VX%(I)=1
3850  graph X%(I)*2-I+1,Y%(I)*2+1,9
3860 END FOR I
3870 HOME=FALSE
3899 END DEFine start

4000 DEFine PROCedure one_move(I%)
4010 graph X%(I%)*2-I%+1,Y%(I%)*2+1,
     board%(X%(I%)*2-I%+1,Y%(I%)*2+1)
4020 X%(I%)=X%(I%)+VX%(I%)
4030 IF X%(I%)>9 THEN
4040  X%(I%)=9
4050  Y%(I%)=Y%(I%)-1
4060  VX%(I%)=-VX%(I%)
4070 END IF
4080 IF X%(I%)<1 THEN
4090  X%(I%)=1
4100  Y%(I%)=Y%(I%)-1
4110  VX%(I%)=-VX%(I%)
4120 END IF
4130 IF board%(X%(I%)*2+1,Y%(I%)*2+1)=0 OR
     board%(X%(I%)*2+1,Y%(I%)*2+1)=11 THEN
4140  SNAKE=TRUE
4150 ELSE
4160  SNAKE=FALSE
4170 END IF
4180 IF board%(X%(I%)*2,Y%(I%)*2)=8 OR
     board%(X%(I%)*2,Y%(I%)*2)=19 THEN
4190  LADDER=TRUE
4200 ELSE
```

```
4210  LADDER=FALSE
4220 END IF
4230 p_man X%(I%)*2-I%+1,Y%(I%)*2+1
4299 END DEFine one_move

4500 DEFine PROCedure p_man(X%,Y%)
4505 LOCal S%
4506 S%=9
4510 IF ((X%+1)/2 MOD 2)=((Y%+1)/2 MOD 2) THEN
     S%=S%+11
4520 graph X%,Y%,S%
4599 END DEFine p_man

5000 DEFine PROCedure throw(I%)
5010 IF DICE THEN
5020  dice1
5030 ELSE
5040  dice2
5050 END IF
5060 REPeat mve
5070  one_move I%
5080  BEEP .1,-10*I%
5090  PAUSE 2
5100  BEEP
5110  M%=M%-1
5120  IF M%=0 THEN EXIT mve
5130  HOME=(X%(I%)=1 AND Y%(I%)=1)
5140  IF HOME THEN EXIT mve
5150 END REPeat mve
5160 IF SNAKE AND LADDER THEN choose
5170 IF SNAKE THEN updown I%,1,3
5180 IF LADDER THEN updown I%,-1,4
5190 HOME=(X%(I%)=1 AND Y%(I%)=1)
5199 END DEFine throw

5200 DEFine PROCedure choose
5210 IF RND>.5 THEN
5220  LADDER=FALSE
5230 ELSE
5240  SNAKE=FALSE
5299 END DEFine choose

5500 DEFine PROCedure dice1
5510 AT 21,2
5520 PRINT "It's your throw ";N$(I%);"     "
5530 PRINT "-";N$(I%)
5550 REPeat DO_DICE
5555  AT 22,2
5560  INPUT "What did you get ";M%
5570  IF M%>0 AND M%<7 THEN EXIT DO_DICE
```

```
5580 END REPeat DO_DICE
5599 END DEFine dice1

5600 DEFine PROCedure dice2
5610 ROLL%=RND(5 TO 15)
5620 FOR Q=25 TO 27
5630  FOR R=25 TO 27
5640   graph Q,R,10
5650  END FOR R
5660 END FOR Q
5670 graph 26,26,22
5680 M=1
5690 AT 21,3:PRINT "It's your throw"
5700 PRINT "  -";N$(I%);"          "
5710 PRINT "  PRESS ANY KEY"
5720 PAUSE
5725 graph 26,26,10
5730 FOR Q=1 TO ROLL%
5735 FOR C=22,10
5736 IF C=10 THEN PAUSE Q*2
5737 IF C=10 AND Q=ROLL% THEN EXIT Q
5745  C%=C
5750  SELect ON M
5760   ON M=1
5770    one C%
5780   ON M=2
5790    two C%
5800   ON M=3
5810    one C%
5820    two C%
5840   ON M=4
5850    two C%
5860    four C%
5870   ON M=5
5880    one C%
5890    two C%
5900    four C%
5910   ON M=6
5920    two C%
5930    four C%
5940    six C%
5950  END SELect
5955 END FOR C
5960  M=RND(1 TO 6)
5980 END FOR Q
5990 M%=M
5999 END DEFine dice2

6000 DEFine PROCedure one(C%)
6010 graph 26,26,C%
```

```
6099 END DEFine one

6100 DEFine PROCedure two(C%)
6110 graph 25,25,C%
6120 graph 27,27,C%
6199 END DEFine two

6200 DEFine PROCedure four(C%)
6210 graph 25,27,C%
6220 graph 27,25,C%
6299 END DEFine four

6300 DEFine PROCedure six(C%)
6310 graph 25,26,C%
6320 graph 27,26,C%
6399 END DEFine six

6500 DEFine PROCedure updown(I%,D%,C%)
6510 LOCal A%,B%
6520 graph X%(I%)*2-I%+1,Y%(I%)*2+1,
     board%(X%(I%)*2-I%+1,Y%(I%)*2+1)
6530 A%=X%(I%)*2+1
6540 B%=Y%(I%)*2+1
6550 REPeat slide
6560  p_man A%,B%
6570  BEEP .1,2*B%
6580  PAUSE 2
6590  BEEP
6600  graph A%,B%,board%(A%,B%)
6605 A%=A%-1:B%=B%+D%
6610  IF board%(A%,B%)=C% OR
      board%(A%,B%)=C%+11 THEN EXIT slide
6620 END REPeat slide
6630 A%=A%+1:B%=B%-D%
6640 graph A%,B%,board%(A%,B%)
6650 X%(I%)=A% DIV 2-1
6660 Y%(I%)=B% DIV 2-1
6670 IF D%=1 THEN
6680  Y%(I%)=Y%(I%)+2
6690  X%(I%)=X%(I%)+1
6700 END IF
6705 FOR I=0 TO 1
6710  p_man X%(I)*2-I+1,Y%(I)*2+1
6715 END FOR I
6720 IF (Y%(I%) DIV 2)*2=Y%(I%) THEN
6730  VX%(I%)=1
6740 ELSE
6750  VX%(I%)=-1
6760 END IF
6799 END DEFine updown
```

```
8000 DEFine PROCedure title
8005 CLS
8006 RANDOMISE
8010 AT 4,14:PRINT "and"
8020 sl
8030 AT 12,3
8040 PRINT "This is a game for two players"
8050 PRINT "  The first player to reach the top"
8060 PRINT "  left of the board wins!!"
8070 AT 21,2
8080 REPeat real
8090  INPUT "Do you want to use a real dice? ";A$
8100  IF A$="Y" OR A$="N" THEN EXIT real
8110 END REPeat real
8120 IF A$="Y" THEN
8130  DICE=TRUE
8140 ELSE
8150  DICE=FALSE
8160 END IF
8170 AT 22,3
8180 INPUT "What is the first players name ?";N$(0)
8190 INPUT "   What is the second players name ?";
     N$(1)
8199 END DEFine title

8200 DEFine PROCedure sl
8210 X1%=1
8220 X2%=16
8230 FOR Y=5 TO 2 STEP -1
8240  graph X1%,Y,2
8250  graph X1%+1,Y,1
8260  graph X1%+3,Y,2
8270  graph X1%+4,Y,1
8280  graph X2%,Y,7
8290  graph X2%+1,Y,
8300  graph X2%+2,Y,6
8310  graph X2%+4,Y,7
8320  graph X2%+5,Y,5
8330  graph X2%+6,Y,6
8340  X2%=X2%-1
8350  X1%=X1%+1
8355 END FOR Y
8360 graph 5,1,2
8370 graph 6,1,0
8380 graph 8,1,2
8390 graph 9,1,0
8400 graph 13,1,4
8410 graph 14,1,6
8420 graph 17,1,4
8430 graph 18,1,6
```

```
8440 graph 1,6,3
8450 graph 4,6,3
8460 graph 17,6,7
8470 graph 18,6,8
8480 graph 21,6,7
8490 graph 22,6,8
8499 END DEFine sl

8500 DEFine PROCedure end_game
8510 AT 21,2
8520 PRINT "You win ";N$(I%);"          "
8530 REPeat an_game
8540  INPUT "  Another game Y/N ?";A$
8550  IF A$="Y" OR A$="N" THEN EXIT an_game
8560 END REPeat an_game
8570 IF A$="Y" THEN
8580  AGAIN=TRUE
8590 ELSE
8600  AGAIN=FALSE
8610 END IF
8699 END DEFine end_game

9000 DEFine FuNction clash(X%,Y%,L%,D%)
9010 LOCal T%,A%,B%
9015 T%=FALSE
9020 E%=2*(X%+D%*L%)+D%
9030 A%=X%*2-D%
9040 B%=Y%*2-1
9050 REPeat sq
9060  FOR Q=-1 TO 1
9070   IF board%(A%+Q+1,B%+1)<>10 AND
       board%(A%+Q+1,B%+1)<>21 THEN T%=TRUE
9090  END FOR Q
9100  A%=A%+D%
9110  B%=B%+1
9120  IF A%=E% OR T%=TRUE THEN EXIT sq
9130 END REPeat sq
9140 RETurn T%
9199 END DEFine clash
```

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. Sometimes the arrangement of ladders and snakes is such that the man reaches the final position and sometimes the man 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 left-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 REMark Snakes and Ladders
 15 PAPER 1
 20 WINDOW 512,256,0,0
 30 init_graph
 40 REPeat game
 50  title
 60  p_sandl
 70  start
 90  REPeat thrw
100   throw 1
110   IF HOME THEN EXIT thrw
130  END REPeat thrw
140  end_game
150  IF AGAIN=FALSE THEN EXIT game
160 END REPeat game
199 STOP
```

Procedure *start* has to be modified to print only one man character and to zero the time. This involves changing line 3810 and adding line 3880:

```
3810 FOR I=1 TO 1

3880 SDATE 2000,1,1,0,0,0
```

One line also has to be changed in *updown*:

```
6705 FOR I=1 TO 1
```

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

```
5000 DEFine PROCedure throw(I%)
5010 p_time
5020 one_move I%
```

```
5030 BEEP .1,-10*I%
5040 PAUSE 2
5050 BEEP
5060 HOME=(X%(I%)=1 AND Y%(I%)=1)
5070 REPeat keyloop
5080 p_time
5090 IF NOT(keypress) OR SNAKE OR LADDER THEN
     EXIT keyloop
5100 END REPeat keyloop
5110 IF SNAKE AND KEYROW(3)<>8 THEN updown I%,1,3
5120 IF LADDER AND KEYROW(4)=1 THEN updown I%,-1,4
5130 HOME=(X%(I%)=1 AND Y%(I%)=1)
5199 END DEFine throw
```

Now, *throw* only calls *one_move* once each time through the animation loop and calls *p_time* to print the current time. The only other modifications are the addition of lines 5070 to 5100, 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 5110 and 5120 which result in the man sliding down a snake if 'S' is *not* pressed and climbing a ladder only if 'L' *is* pressed. Notice that due to the line number changes it is better to type this procedure in again rather than try to modify the existing version of *throw*.

Procedure *p_time* is a new but very short procedure that prints the time at the bottom of the screen:

```
4600 DEFine PROCedure p_time
4610 time$=DATE$
4620 sec=time$(17)*60
4630 sec=sec+time$(19 TO 20)
4635 AT 22,5
4650 PRINT "TIME=";sec
4699 END DEFine p_time
```

The changes to *title* are simple but extensive and this makes it worth listing the entire new version of the procedure:

```
8000 DEFine PROCedure title
8005 CLS
8006 RANDOMISE
8010 AT 4,14:PRINT "and"
8020 s1
8030 AT 12,3
8040 PRINT "Your must try to get you man to"
8050 PRINT "  the top lefthand corner of the"
8060 PRINT "  board as quickly as possible"
8062 PRINT "  You must avoid snakes heads by"
8064 PRINT "  pressing S and climb ladders by"
8066 PRINT "  pressing L"
```

```
8070 AT 21,2
8080 PRINT "Press any key to start"
8090 PAUSE
8199 END DEFine title
```

The same is true of *end_game*:

```
8500 DEFine PROCedure end_game
8510 AT 22,2
8520 PRINT "You took ";sec;" seconds"
8530 REPeat an_game
8540  INPUT "  Another game Y/N ?";A$
8550  IF A$="Y" OR A$="N" THEN EXIT an_game
8560 END REPeat an_game
8570 IF A$="Y" THEN
8580  AGAIN=TRUE
8590 ELSE
8600  AGAIN=FALSE
8610 END IF
8699 END DEFine end_game
```

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 sort of 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 that large programs are written. However, to really get the most from what you have learned it is essential that you gain practical experience by first 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 own 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 which is described at greater length in *The Complete Programmer* by Mike James (Granada, 1983) – but this is just part of the story. You could say that structured programming and stepwise refinement are the 'scientific side' of the craft of programming. Without their application programming is hard work. However, 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 first have to know what to value and you have to finish the programs that you start.

Games and problem solving

If you look back to the earlier chapters you will find that most of the programs started from a simple idea which, implemented in BASIC, did not produce a 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 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 it will be frustrating rather than exciting.

If a game doesn't live up to your expectations then do not immediately 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 sometimes be difficult to track down. It can be due to the length of time that the animation loop takes to execute but it can just as easily be due to not inspecting the keyboard often 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 isn't to speed up the animation loop but 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 the game really does depend on the skill of the player, setting a short time limit either has no effect or it 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 fulfils 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.

Finishing

Once you have a game that is fun to play then 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 misunderstand it and so the fault is theirs; if they misunderstand it, you didn't tell them enough about it or it is too complicated and in any case a well-written program shouldn't crash!

All the games in this book have been 'crash-proofed' in the sense that it should be impossible to input values or control the games in such a way as to make them 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 value 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 and the foot 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!

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 *The Complete Programmer*, 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 problem that causes 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 successful 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 be small ones!

There is a particular problem associated with testing games programs that doesn't generally occur with other types of program, and that is 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. 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. Firstly, 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 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 be made to happen 'on request' before trying to fix the bug. In the case of Snakes and Ladders, once a board had 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 then you will 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 of programming is to be found. However, producing 'user-friendly' programs is just as important and just as skilful. In some senses it is the most skilful 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 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 his program in such a clumsy way that it causes 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 the games in this book are based on animation. The main feature of all such games is that the player has control over the motion of some object on the screen. Although you can create new animation games by thinking up fresh themes and objectives, the only real difference between the games lies in the way 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 how the player will control the moving object and what new element of skill is required.

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 players' 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, includes 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 ingredients in different ways. This is yet another reason for always writing programs using procedures. If a game has been written using procedures then 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 QL

You may be surprised that the excellent high resolution graphics capabilities of the QL were largely ignored by the games in this book. This is, however, easy to explain in that the main use of high resolution graphics in games is to create backgrounds. Most animation games are much easier to implement using user-defined graphics which are not initially provided by the QL's Super BASIC. So, while the QL's standard range of graphics facilities is well suited for use in commercial and technical applications, they are not immediately capable of producing the effects required for animated games. However, as we hope we have demonstrated throughout this book, it is very easy to remedy this deficiency and add the necessary procedures which can then be used in the same way as the QL's own built-in functions. This ability to write extensions to Super BASIC is one feature that makes the QL a particularly flexible and versatile machine. When writing your own collection of programs you will find it extremely useful that you can build up a library of your own procedures which can then be merged into subsequent programs.

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. Your 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 created and finished something that other people can play.

EXCITING GAMES ROUTINES – AND HOW TO MAKE THEM EVEN BETTER!

This book of high quality games for the Sinclair QL shows you how to develop your own games as you learn the sophisticated techniques professional games programmers use. You will also pick up, the easy way, the coveted skill of solving programming problems as they arise – while becoming increasingly familiar with the QL and its many facilities.

This book provides well-structured programs so that each procedure or 'module' performs a discrete and identifiable task which enables variations on the 'core' program to be written by substituting some new modules while keeping others as they are. You are also shown how to customise your own programs very easily, and in so doing improve your programming skills by leaps and bounds.

The book includes adaptations of some of the most popular arcade games so that you can create your own favourite versions.

The Authors
Kay Ewbank, Mike James and S. M. Gee have collaborated on many programming projects, and between them have written several successful books. They are also regular contributors to *Electronics and Computing Monthly* and other computer magazines.

More books for QL users

QL COMPUTING
Ian Sinclair
0 246 12595 0

QL SUPERBASIC
A. A. Berk
0 246 12596 9

PROFESSIONAL AND BUSINESS USES OF THE QL
Colin Lewis
0 00 383005 5

Front cover illustration by Jeff Ridge

COLLINS
Printed in Great Britain
0 00 383066 7

£7.95 net