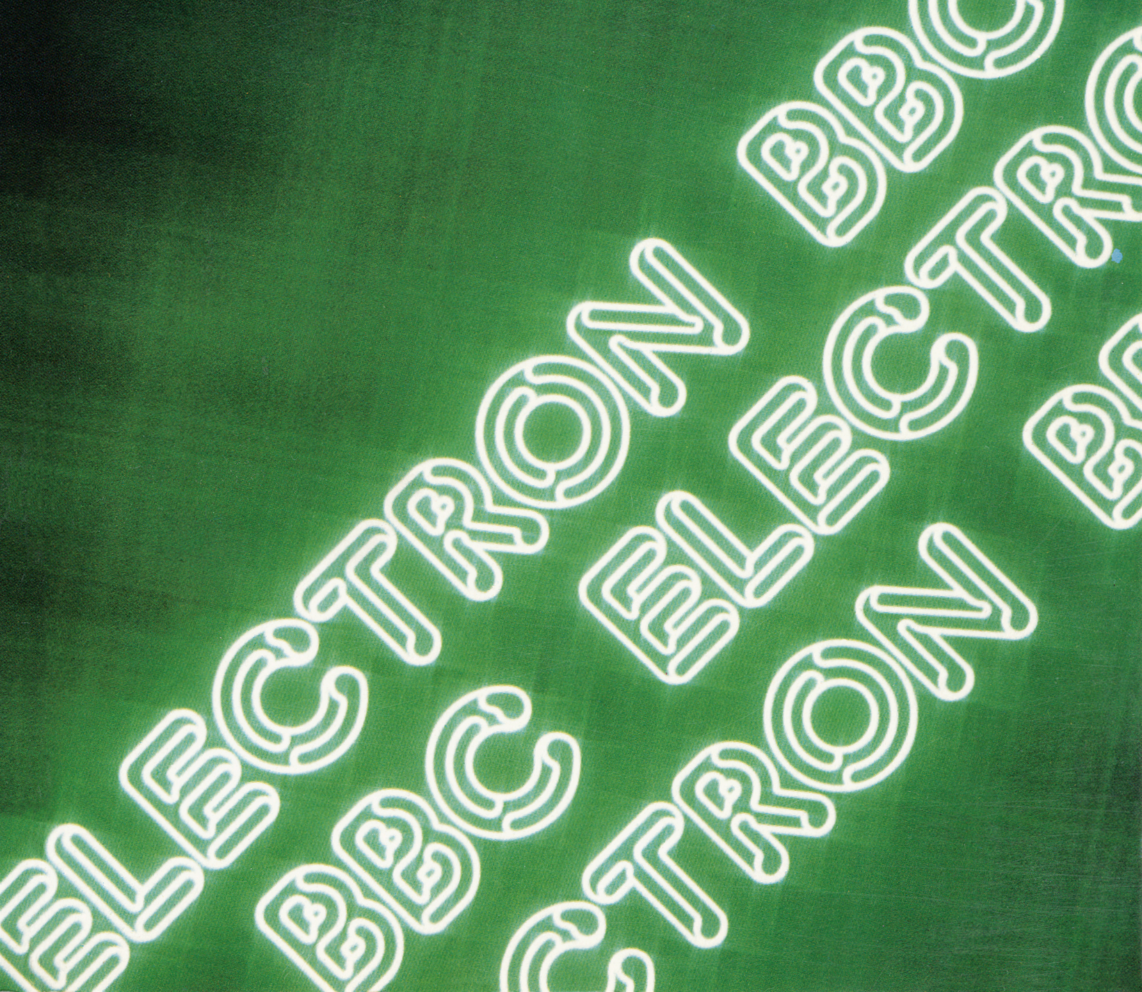
INTELLIGENT ADVENTURES FOR THE ELECTRON AND BBC MICROCOMPUTERS

Noel Williams



INTELLIGENT ADVENTURES FOR THE ELECTRON AND BBC MICROCOMPUTERS

Intelligent Adventures for the Electron and BBC Microcomputers

Noel Williams

McGRAW-HILL Book Company (UK) Limited

London · New York · St Louis · San Francisco · Auckland · Bogotá · Guatemala
Hamburg · Johannesburg · Lisbon · Madrid · Mexico · Montreal · New Delhi
Panama · Paris · San Juan · Sâo Paulo · Singapore · Sydney · Tokyo · Toronto

Published by
McGRAW-HILL Book Company (UK) Limited
MAIDENHEAD · BERKSHIRE · ENGLAND

British Library Cataloguing in Publication Data
Williams, Noel
Intelligent adventures for the Electron and BBC Microcomputers.
1. Computer games 2. Microcomputers
I. Title
794.8′028′5404 GV1469.2

ISBN 0-07-084749-5

Library of Congress Cataloging in Publication Data
Williams, Noel
Intelligent adventures for the Electron and BBC microcomputers

1. Electron Microcomputer – Programming. 2. BBC Microcomputer – Programming. 3. Computer games.
I. Title.
QA76.8.E38W55 1985 001.64′2 84-17134
ISBN 0-07-084749-5

12345 CUP 8654

Typeset by Oxprint Ltd, Oxford
Printed in Great Britain at the University Press, Cambridge

For my mother (who is ultimately responsible)

CONTENTS

PREFACE

You obviously know something about intelligence because you have just done a highly intelligent thing—you began to read this book. If I was trying to sell it to you (I'm not, of course) I could say that you will demonstrate even greater intelligence by reading it from cover to cover. But this book is not about the intelligence of you, the reader. It is about the intelligence of computers and the intelligence of people who play computer games. On the one hand it aims to show some of the basics of artificial intelligence and how it can be used in games. On the other it shows how you can go about writing games which require some intelligence to play. Of course, the two things go together.

Therefore if you want a book of one hundred and forty-four listings that can be played in your sleep or you want to know how to write arcade games that move so fast no-one can see their scintillating graphics, you have probably picked up the wrong book. The listings in this book are substantial, making extensive use of many of the facilities of the Electron and BBC micros, but pay little attention to aliens. There is one example of a graphic game but all the other games require thought, and time, to play.

The most common example of the thinking man's computer game is an adventure. You will find four chapters on adventure which take you through all the processes of designing and writing your own adventure to the complexities of manipulating bytes and data compaction, sentence processing, and semantic databases. This culminates in an eighty-one room adventure which uses just about every byte of the Electron's RAM. There are other types of game which require intelligence to play (and to program) so there is a chapter on writing an abstract board game and a chapter on simulation, both illustrated by substantial programs—Dilemma is an original computer version of a board game; Mernar Keep is a simulation cum wargame.

You will also find advice on designing games, on programming with the Acorn machines, and on how to develop artificial intelligence routines, as well as many ideas on programs you yourself can develop. There is a game that learns, two programs that create new ideas, and a set of programs to take the slog out of producing your own adventures, as well as many illustrative routines to incorporate in your own work.

This might sound too good to be true—one book which introduces artificial intelligence, gives all the basics of adventure design, describes the considerations of game design, develops your knowledge of the BBC and Electron, and gives three major programs which might cost the price of the book on their own. What has been missed out?

I have assumed a beginner's knowledge of BBC BASIC and one or other of the micros that use it, so there is no coaching on elementary BASIC (though all the more unusual or advanced jargon is explained). I have kept discussion of graphics in the background, giving less than a chapter to one of Acorn's best features, and I have hardly mentioned sound. I have had to keep the text as brief as possible and the examples simple—so that the spelling checker in the adventure can only make suggestions about the player's intended input and the sentence processor (or parser) copes with a limited range of word types. The vocabulary of the adventure could have been larger than fifty-one verbs and eighty-seven nouns if there had been room to introduce machine code, and the Dilemma program could have played a virtually unbeatable game (as it is, it wins about sixty per cent of the time).

In other words, the book aims to illustrate an enormous and exciting field and does so through extended example, rather than by trying to show everything and do everything. Hopefully you will find an approach to programming and to writing and playing games which is a little more rewarding than the continual bombardment of aliens, and through it you will understand the elements of what is likely to prove the next major stage in the development of computers—the creation of a thinking machine.

One thing that will never be developed is a machine that could replace the love and care of my wife, Carrol, without whom this book could never have been written.

1 AN INTRODUCTION TO INTELLIGENT GAMES

This book is about intelligent games, not only for the BBC/Electron (for which you will find four major listings in this book and several minor ones) but for any micro with enough memory (a minimum of 16K). There is plenty in the book which is specific to the BBC/Electron which users of other micros would need to adapt, but all the principles of design, coding, programming, and presentation discussed in this book will apply to games design whatever micro you own.

What do I mean by 'intelligent games'? Two things. Firstly, there are games which require some intelligence to play. For all the micros currently on the market there are innumerable games of the arcade style, with fast moving animated graphics and devastating sound. These games are difficult and fun to play but generally they do not require much intelligence. Instead, you need quick reactions, a good eye, and a good knowledge of the game. These are typically combat games—shooting down aliens or defending your missile base—or travelling games—negotiating mazes at high speed or climbing up and down ladders and ramps. If such games were real then the real skills involved would be most appropriate to a soldier—skills of dexterity, of accuracy, of avoiding panic, of manipulation, and of concentration. However, this book is concerned more with the skills of the officer, the person responsible for organizing and marshalling the many skills of subordinates. The officer may well lack the ability of those under his or her command, but requires extra skills. These are skills of planning, of organization, of logistics, of tactics and strategy, of psychology, of language, and of accurate guesswork. The games in this book require those kinds of skills and the book describes approaches to programming games which are primarily of this kind. Although there is some discussion of graphics and one program demonstrates their uses, you will not find many tips on how to write fast arcade games.

As more and more people learn about micros and pass beyond that early fascination with the magnificent arcade games which are available, so the interest in intelligent games is widening. Many people interested in programming seem to have the sort of mind that likes games playing, whether it is solving puzzles, playing abstract board games like Chess and Othello, role-playing games of fantasy and adventure, or wargames and simulations. These games not only demand logic and intelligence but they teach it. In particular, educationalists have been quick to realize that not only can the micro teach people about subjects like geography, mathematics, and history, it can teach them a great deal about how to think, how to plan, and how to solve problems. But this book is not really for the teacher; it is for fun. There are still too few games around which demand intelligence in players. Hopefully, when you have read this, you will be able to do something to rectify that.

The second meaning that 'intelligence' has in this book is the sense of 'machine intelligence'. Perhaps you have come across this phrase in computing magazines or perhaps have heard of the study of artificial intelligence. Called AI for short, this is a branch of computing which uses much research in psychology and linguistics. It aims to learn about human intelligence by discovering what it would take to make a machine intelligent. If you think about it for a moment you will see that if we want to write a game which requires intelligence to play then the game must have some intelligence in its design. Either it has to be designed by hyper-intelligent alien programmers with convoluted minds who are capable of designing the microtorture known as adventure games (you will see how to become one of these later in the book) or the game itself must have some intelligence. So either the program must be intelligently designed in order to cope with all the inputs and strategies of a player or it must use some intelligence itself to cope with the player as the game goes on. In work on AI, researchers have tried to discover what a program has to do to behave like a human being so it can respond to intelligent actions by a player.

Of course this research has not been designed with the main idea of making better games. It has offshoots into robotics, medical and psychological research, linguistics and machine processing of text, weapons guidance systems, improved computer design, and many other fields. The upshot is that if a piece of research in AI can make a machine act more intelligently (or seem to act more intelligently) then that idea can be used in games programs to make them play more intelligently.

This book is not going to debate the implications or effects of AI. Perhaps the machines with such intelligence are really thinking, in some sense, and perhaps one day we may want to call them 'alive', but at the moment AI is in its infancy and there are many problems to solve before we will see a truly intelligent machine. However, for our purposes, if we can get a machine to do what a human being could or would do in similar circumstances we can regard that machine as 'intelligent', because what we want is a machine that plays like a human being. Or perhaps we want one that plays rather better than a human being. At any rate, the program must act in a sensible and organized way.

There is no point in pretending that any of the games in this book have a great deal of built-in intelligence. The field is fraught with problems and many of them do not yet seem soluble on a microcomputer—if at all. What we are going to do in the book is explore the ways that a game can be made interesting from a player's point of view, particularly by using approaches derived from AI. However, each game only uses a small sample of the possible techniques, and relatively simple techniques at that. The idea is to stimulate your own thinking on game design so that you can produce original ideas, and to do this by guiding you through various stages of game design and different kinds of game so that you can design your own games without problems. The kinds of games we are interested in are simulation games, wargames, adventure games, and abstract games.

One problem with games of this kind is that they tend to be large. They use a great deal of RAM (i.e., Random Access Memory, the memory that your programs use in the micro). There is a direct trade-off between the interest and intelligence of the program and its length, so most games which require intelligent play or apply AI techniques tend to leave little space for the trimmings of graphics and sound. However, because this book is as much about game design as about AI, and because there are interesting things which can be done with graphics in intelligent games, there is a chapter on them in this book. Remember, though, that if you want to design your own intelligent adventure or complex strategic game, even with a 32K BBC/Electron, you are unlikely to have much room for stunning graphics. There is almost always a trade-off between the visual brilliance of the graphics and the internal brilliance of a complex game.

The approach used in this book is that known as modular or structured programming. This means that a program is thought of as a structure made up of a series of modules. Each module is self-contained, generally as a subroutine (or, in the language we will be using, BBC BASIC, a PROCedure), so different modules can be defined in different ways to give different programs. In this way many different games can be written with only very small changes—the same modules can be used again and again with minor modifications. Modular programming has a number of other advantages. It encourages programming habits which make error-checking and debugging easier; it encourages a sensible and logical approach to program design; it generally makes listings easier to read and understand; it makes the transition to structured languages such as Pascal much easier for those who want to learn a new language; and it makes program design both easier and more elegant. Its only real disadvantages are that it requires some self-discipline from the programmer and programs may sometimes use more memory or take more time than unstructured equivalents, but these possible losses are well worth the gains. If you only want to write one game which saves as much memory as possible and runs as fast as possible, you have little to gain from structured programming. However, if you plan to write several such games, or to build a library of routines for such games, the initial work will be amply repaid later on.

The main purpose of this book is to help you in game programming and, through games, to help you learn more about programming principles and design. We will look at the different components of games, such as screen display, logic, idea generation, user friendliness, and playability and how these can be programmed. We will look at the advantages and disadvantages of different techniques, such as ways of storing data, ways of interlinking variables, and ways of accessing information. We will look at some of the more common jargon words such as ‘random access’ and ‘menu’, as well as ‘artificial intelligence’, to explain them in a straightforward context. We will also look at some of the most useful or unusual features of BBC BASIC. We will do all this by writing our own games as we go along.

The book is organized in the following way. We begin by looking at the different components in the game, with a brief sketch of different kinds of game. Then comes the main part of the book, eight chapters of programming and design techniques for use on different types of game, beginning with overall design and working through all the major components and working on successively more complex design until we are ready in the last two chapters to develop a full adventure. Most sections give one or more routines you can incorporate in your own programs, usually in the form of procedures, and we build up four complete games: Pantry Panic, Mernar Keep, Dilemma, and The Opal Lily. Between them these four games illustrate most of the essential principles discussed in each chapter as well as containing many routines which can be used in other games. In particular you will learn the processes for manipulating data which are normally called ‘writing an adventure game’.

All the programs in this book have been developed on a BBC micro and tested on an Electron. They will run on both machines, though BBC owners with the 0.1 operating system may discover a quirk or two. As the Electron and BBC hardly differ in their BASICs, operating systems, or memory maps there is unlikely to be any problem running an Electron program on the BBC, but the reverse is not the case because the elder brother has Mode 7, additional interfaces (and so the facilities for controlling them), and a different sound system. You will find that sound is one area that has been neglected by this book, partly because it is not very useful when considering the topic of intelligent games, but mainly because of the differences between the systems. I do not wish to waste the time of one set of readers by explaining all the features of a micro they do not own! However, in one or two cases there are some features of difference between the machines which are important, e.g., Electron owners have the programming advantage of single-key entry for many BASIC keywords whereas to achieve the same thing BBC owners must program function keys. However, the latter have a set of dedicated cursor control keys which can be used for games whereas the former have to use keys on the normal keypad. In the one or two cases where differences such as keyboard layout might affect the nature of a design I have given alternative suggestions.

Having been designed with Electron owners in mind none of the programs use Mode 7. BBC owners will therefore find that they have some advantage over users of the Electron where an expansion or adaptation of the programs is concerned because both Mernar Keep and The Opal Lily, as well as several other lesser routines, can easily be converted to run in Mode 7, which releases an additional 7K for extra programming. The only necessary alteration of the former would be to redesign the map using Teletext graphics. For the adventure more work may be needed as the large block of memory which holds the data for the objects and the map as well as the dictionary of allowed words will have to be moved in order to take advantage of the extra RAM. This means that all lines using this block of memory will need their addresses changed, but how to do this is discussed in the relevant chapter.

2 DESIGNING AND PLANNING A GAME

2.1 Choice

Playing a game, using strategy and tactics, is a question of making a series of choices. Each choice depends on previous choices, so if the player makes the wrong ones, he or she loses. Writing a game is a process of creating a set of possible choices for the player to make. The fundamental BASIC command for a game is thus IF . . . THEN . . . ELSE . . . If the player chooses to do (a) then (b) will happen, but if (x) is chosen then the result will be (y).

Therefore if we want to write a good game we must do three things:

1. Offer the player some interesting or testing choices.
2. Organize those choices in a coherent way, so that the conditional choices fit together sensibly.
3. Make the interrelations between the choices complex enough to be worth while to solve but simple enough to be solvable.

We will look at the programming implications of these later, but the simple lesson is that the game should play in a straightforward manner although that apparent simplicity should depend on complex hidden balances in the programming. You will see this particularly in the game Mernar Keep in Chapter 4. This game is really a large collection of variables fitted together in such a way that any action by the player which affects one variable will also have an effect on some of the other variables. The player can choose any variable to increase or decrese, such as increasing the size of the army or reducing the amount of forest on lands, but it must be remembered that each of these choices may affect other variables and therefore prevent other choices in the future. For example, if the player recruits more men at arms in Mernar Keep there will have to be fewer peasants. This means that there may not be a large enough work force to plant the crops, which may in turn lead to a poor harvest and consequently to starvation for the troops just recruited.

Games can depend on many kinds of choice. Initially the player may be able to configure the game or some of its parameters, e.g., by choosing the level of play or the type of character to play in an adventure; there may be a possibility of choosing various actions in different situations (e.g., running from, talking to, or fighting an encountered dragon); the player might have to choose when to use limited resources (e.g., when to commit the cavalry reserve in a wargame) and even when to end the game (as with the save facility incorporated in many puzzle adventure games).

To be a little pedantic for a moment, we could call a player's strategy a series of choices aimed at optimizing rewards in a game. So the choices that make a game interesting depend on the kind of rewards the game offers. At the most abstract level a game offers points. The better the player, the higher the score, i.e., the better the strategy and hence the series of choices made, the more points will be obtained.

We could have a very simple game in which the player has to guess a number between one and nine chosen by the computer. The player selects a number, makes a guess, and the computer tells the player if its chosen number is higher or lower than the guess. The player starts with a score of 100. Every time a guess is made, 10 points are lost, so the better the guesses the higher the final score. Initially there are three major choices—three possible strategies. Players can guess numbers randomly; or they can start at one end of the scale and work up (or down) till they hit the right number; or they can use the following routine:

1. GUESS 5
2. IF NUMBER IS LOWER THAN 5 CHOOSE 3 ELSE CHOOSE 7
3. IF 7 WAS CHOSEN AND THE NUMBER IS LOWER THEN THE ANSWER IS 6
4. IF 7 WAS CHOSEN AND THE ANSWER IS HIGHER CHOOSE 9
5. IF 9 IS CHOSEN AND THE NUMBER IS LOWER THEN THE ANSWER IS 8
6. IF 3 WAS CHOSEN AND THE NUMBER IS HIGHER THEN THE ANSWER IS 4
7. IF 3 WAS CHOSEN AND THE ANSWER IS LOWER THEN CHOOSE 1
8. IF 1 WAS CHOSEN AND THE ANSWER IS HIGHER THEN THE ANSWER IS 2

If players choose the random strategy they may take up to eight guesses to get the correct answer and should average about five. If they choose the second strategy the same is probably true. For the third strategy, however, the number of guesses is never greater than four so players will always score at least 60 points, and will probably average around 80. Obviously the best strategy is the third strategy because at each stage in the game the best choice of all the available choices is made, i.e., the choice which reduces the number of possible future guesses to the fewest.

In order to get an idea of how balanced your game is while you are designing it use a points system against all the options as a rough measure of the difficulty, reward, and balance built into the game. You do not need to incorporate that system into the actual game itself, but it can be a great help in designing to enable you to know how easy or how complex your game is. For example, if you think that each problem in your game is so easy to solve that it is only worth one point you would be surprised during playtesting if you found that the way you had put all the problems together meant it took half an hour to score two points. There would have to be a design flaw for something like that to happen. Perhaps the problems are harder than you thought, perhaps the rewards of the game are not great enough, perhaps the overall structure is at fault, or perhaps more clues or instructions are needed.

Points are not the only kind of rewards, however. If we think of our compulsive games player (let us call him 'Igor', short for Intelligent Game Owner and Runner) as being a playing machine running a program called 'I'll enjoy this if it kills me', we can regard REWARD as one of the variables in that program which determine whether it is successfully run or not. If REWARD falls below 1 then the program will END. But values can be given to that variable by a host of functions, only one of which is the 'increase points' function.

There are typically two kinds of rewarding function which Igor will respond to, namely local rewards and global rewards. A local reward is one which temporarily increases REWARD as a result of an action just completed, but which has no permanent effect on the rest of the game. For example, our hero may slay the Great Green Slime Beast of Trag. He will feel rewarded the first time he achieves it, but if he gets no points, finds neither treasure nor clues, does not increase in skill, etc., he will quickly forget it. Some games are made up of a series of such local rewards—there is no cumulative REWARD, just a series of temporary increases in its value. For such games to work they must keep the successes close together, so that there is no time for REWARD to fall below 1, and each victory must be different or new, or REWARD will not be incremented. This is part to the philosophy of increasing the level of difficulty in arcade games. Each time Igor succeeds at one task he is given a more difficult task, so the reward is correspondingly greater.

Global functions which increase REWARD are the basis of good adventures. The player must be given enough local success to keep going from stage to stage, but will only want to complete the game, to keep returning to it, if some successes increase the chance of overall victory. In the simplistic number-guessing game above Igor will continue to play as long as he thinks he is getting nearer to a solution, but if every guess did not affect the chance of a future guess being correct he would rapidly lose interest. And quite right too! What is the point of playing if nothing you do increases your overall chance of winning?

So the typical game has a series of hurdles to be overcome in order to achieve the final victory. Each hurdle jumped will act as a local reward and temporarily increase REWARD. After a while that temporary increase will be lost, but there will still be an overall increase because the player will know that now 'the answer' is much nearer. In an adventure these hurdles may be problems to solve, objects to find, mazes to get through, monsters to defeat, riddles to answer; in an abstract game each new move is a hurdle because of the new configuration of the board; in a strategic game or a wargame the hurdles include all the various stages of planning and acting correctly in a constantly changing situation. A simple problem like 'How do I open the door?' can provide enough reward to keep a player going for several hours if it is known that the answer is on the other side. Similarly, deciding exactly how many camels to load with spears and how many to load with gold can keep Igor coming back again and again if he sees that getting it right will finally enable him to get across the Sahara.

In order to gain overall victory each of the separate problems must be overcome. At any stage in a game therefore, Igor will have two objectives: (1) to solve the particular local problem that faces him at the moment and (2) to solve the overall problem. Obviously anything which helps with the former will count as a local reward for him and increase his enjoyment temporarily, whereas anything contributing to the latter will give a permanent increase in REWARD. Winning in this kind of game is like solving Rubik's Cube. There is pleasure in getting each individual coloured square in place, but that is nothing like the overwhelming smugness that comes from being able to put the lot together, every time.

So in designing our game we must bear in mind that the player should feel, in part at least, to be progressing through a logical series of choices. Random choices do not usually make satisfactory games. It does not matter if some of the elements are random or fully predetermined, but they should seem coherent from Igor's point of view. In an adventure this might be done by creating a storyline which the player progresses through as the main character. In a simulation or wargame this is done by having a setting or scenario which makes sense of the choices. In an abstract game, however, the approach is slightly different. A set of logical rules are constructed and the player has to apply those rules to the best of his or her abilities to achieve some final position. There is usually no ‘realistic’ basis for an abstract game. The pleasure comes from trying to discover the best way to apply the rules rather than from imagining one is solving a real problem of some kind.

2.2 Structures

Almost all games can be reduced to a number of key elements, the relationships between those elements being shown by one simple flow chart. In the rest of this chapter we will look at each of those basic elements and the whole flow chart, so that in the following chapters these general ideas can be turned into a design and then a game. Each of the following chapters takes one or more of these elements and discusses them in detail, exploring some of the different ways they can be handled.

These elements will vary slightly according to the particular machine used (e.g., whether it has colour or high-resolution graphics), but at the most general level games all have each of the following: some form of input/output, made up of a textual component (what is written on the VDU, such as a description of a room and its contents), a visual component (perhaps a graphical picture of that room or a flashing warning), a sound component (magical explosions, beeps, and burps), together with three structures—the game structure (what the rules of play are and how they are to be carried out), the story or puzzle structures (what the player is meant to be doing in the game), and the program structure (how each of the above elements fits together in a way that a computer can understand). The basic model of a game is shown in Fig. 2.1.

These are the seven areas we must consider at the earliest stage in designing our game. Clear and original ideas at this first step will be repaid later on. On the other hand, if we only have ideas on what the game is about without considering how it will look to the player, or if we decide we are going to design a game using speech but we do not have any idea about how it will be played, sooner or later our coding will come to a complete stop, or need large amounts of rewriting, or, if it ever is finished, result in a dull and unplayable game. Of course, you should not plan each of these areas in a way which is completely separate from the others. They are convenient abstractions to aid

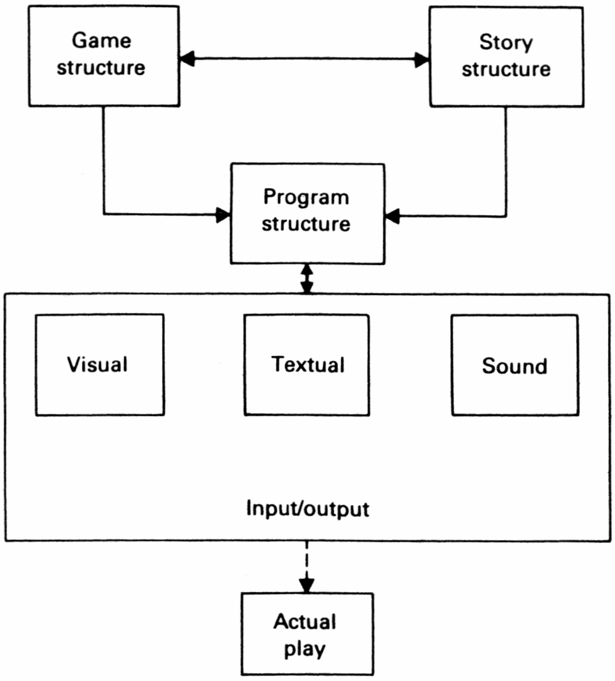


Figure 2.1

design, not commandments that must be stuck to or completely closed categories. Later in this chapter we will look at some design strategies that may be used, but first we will make a brief examination of each of the boxes in Fig. 2.1, the kind of content they may have, and the decisions it is necessary to make in filling them.

2.3 Story structure

Many games begin with an idea for a particular storyline or scenario. You might think that it would be a good idea to have an adventure about climbing down the inside of a volcano to the centre of the earth; or you might like a game in which players sail the Kontiki across the Pacific; or wish there was a game where you could actually talk to characters you find instead of just killing them. However, having had your initial flash of inspiration, you may not know how to turn that into a description which can be coded for the micro. After all, many people have tried writing stories, but there are only a few Flemings and Macleans.

You do not need to be a great novelist to write a story, scenario, or puzzle that can become the basis of a program, especially if it is not an adventure. In the case of the arcade type all you need is a convincing scenario to make the game seem to be about something. For a wargame or simulation a little more thought is needed. For an adventure the story may be the most important element so it pays to devote a little time to it. All you need to remember are two things:

1. A story is a series of linked events in which a character (or group of characters) is changed in some way.

2. An event is made up of a place, a particular time, one or more characters, one or more objects, and some possible actions or consequences.

Now we can draw up our story structure. We write down, either descriptively or as a flow chart, all the events we would like to include and how they might be linked together. This in turn involves deciding on the character or characters, how they might change (e.g., they could become rich, injured, learn to fly, etc.), what places or locations there will be, what period it is set in (the Napoleonic wars, the far future, prehistoric times), what kinds of objects might be found or used (starships, treasure chests, maps, weapons, horses, boats, religious relics), and what kinds of actions can be taken (can the characters be killed, can they talk or write, how can they move from place to place, can objects be carried and if so which, will magic or futuristic devices be included?). You will find at the end of this chapter a list of possible storylines which have not been used much, if at all, in games for you to adapt to your own games. These should be especially useful if you are looking for a new idea for an adventure plot.

We also have to decide on the way or ways the story might end. Will it only end when the main character is killed? Will the character have a task to fulfil or a problem to solve? Will the character have to gather a certain amount of treasure or reach a particular social level? These questions obviously relate to the game structure, because the ways our story or scenario might end determine what counts as winning or losing the game.

To illustrate how we can do this, here is the framework for a simple story. Chief Iron Buffalo (the main character) must lead his tribe to new hunting grounds before winter sets in. The tribe can pass through mountains, forests, and plains (the possible locations) where they may meet wolves, bears, the Long Knives, or settlers (these are the other characters) and may find ponies, buffalo, and a magic tomahawk (objects). They can choose to attack, run from, or talk to other characters, to ride or hunt the ponies or buffalo, to take or leave the tomahawk, and they must eat a certain amount of food each week (possible actions). They might be given extra food, or be shown a short-cut, or be attacked, by other characters (further actions). Possible consequences are that the tribe will starve, or become lost in the wilderness, or all be killed by the white men and wild animals, or revolt against the chief and scalp him, or they may find new hunting grounds (story endings).

Here are all the elements of our story. You could also use the same kind of checklist to provide the background setting for other types of game, though it is not usually necessary to go into so much detail.

Having decided on story details we must then decide how to string them together and how to fit them with the six other basic program areas. For example, what are the chances of encountering settlers in the forest; will there be graphic illustration of any combats; how will the tribe move from place to place?

2.4 Game structure

All games have essentially the same structure— the player makes a move; the consequences of the move are calculated; if the player has won or lost then the game ends; otherwise that player or another player makes another move. We can represent this by using essentially the same control loop (often called a supervisor) to regulate all our games. Its general form would look like Fig. 2.2.

```
9 REM Main Loop

10 win = FALSE: lose = FALSE

20 PROCset_up_variables

30 PROCprint_instructions

40 REPEAT

50   PROCplayer_move

60   PROCconsequence

70   PROCother_move

80 UNTIL win = TRUE OR lose = TRUE
```

Figure 2.2

This same supervisor could be used to control an arcade game, in which case PROCother_move would probably be used to move targets around; or it could be used in a two-player game, in which

PROCother_move would be the second player, or in a game against the computer, such as Chess, in which PROCother_move would be the computer's turn, or in an adventure, in which PROCother_move would be the actions of any characters in the adventure which were not the direct result of the player's action.

In some games, such as adventures and simulation games like Mernar Keep in Chapter 4, there are seldom moves in the traditional sense of, say, a move in Draughts. What happens instead is that the player is periodically presented with a series of choices which represent the kinds of actions the character(s) could take in the type of world represented in the story. These are often presented as a menu (see Chapter 3). The choice made is the 'move' and its consequences will be some effect on the world represented by the program or the persona that the player has assumed in that world. In an adventure the player may choose to DRINK WATER only to be poisoned. In a simulation of governing Britain the player may choose to increase taxes only to find that a general strike results.

The kinds of options, choices, and modifications available in the game depend on the story or puzzle structure we choose, but the basic system remains the same. However, there are obviously variations which can be chosen, otherwise games would all be roughly the same and rather tedious. For example, one of the options may lead immediately to a choice of further options without any modification of the player or the world. In this way the player may be presented with the list of choices:

1. ATTACK
2. HYPERSPACE
3. RESEARCH

If ATTACK is chosen, a second series of choices may immediately be given, each of which is a subcategory of attack:

1. FIRE MISSILE
2. FIRE TORPEDOES
3. RAM OPPONENT

Such increasing specificity can be continued indefinitely, up to the nesting capacity of the micro used, i.e., the limit on the number of embedded PROCedures, REPEAT . . . UNTIL loops, or FOR . . . NEXT loops. Many micros only allow a limited number of embedded subroutines which would restrict this kind of nested specificity.

A second game variation involves the effect which the choice has on the player rather than on the character or the world. For example, a request for further instructions or to print the character's current status is a part of the game structure which has no effect on the game. However, it would be perfectly possible for such choices to affect the game. So every time a player asks for instructions the intelligence variable of the character could be reduced. This kind of modification makes playing the game more skilful, and that is a primary consideration in designing a game. An enjoyable game is one that is difficult enough to be taxing yet easy enough to understand. Abstract games such as Othello and Chess are very popular for just this reason—they are simple to learn, but difficult to play well. This is why the Dilemma game has been included in this book (in Chapter 5). The concept is very easy to learn but good play is quite taxing. Similarly, the game structure of an adventure should make play easy but good play difficult. You will find that some of the programming in both Mernar Keep and The Opal Lily is quite complex even though the options available to the player are few and simple. Essentially a game which requires some intelligence to play has to encode a number of intellectual difficulties or puzzles by intricate programming, but it must not make them appear too difficult to players or they will not bother to play the game.

A third variation is thus to alter the possible options and consequences of a player's moves and/or to make them affect each other. In a good game a choice made in the first few moves can have a significant effect on choices available much later on. Part of the game structure should therefore be a description of how choices are related to each other. For example, if, in Chess, you choose to place all your pawns on black squares you make movement easy for your white bishop but difficult for your black. Our game might involve designing a spaceship. If the player chooses a great deal of weaponry, perhaps only a slow ship should be allowed. Alternatively, suppose a player in a fantasy game finds a magic staff. In the short term this may do some good as spells can now be cast, but perhaps it is cursed to destroy its owner in a room with goblins in it, which may be a long-term disadvantage.

In particular, in designing a game structure we should decide on how each operation will function. No choice in a game should automatically result in success. Either the choice should bring success only if made together with other correct choices (e.g., you must choose the black bow but only the white arrow) or there should only be a chance of success represented by a function curve of its probability. Such curves are the heart of many games. For example, suppose our game involved the character moving up the social scale from peasant to king and each turn represented a year in his life. As he goes higher up the social ladder his knowledge increases. We would represent this by the curve in Fig. 2.3. However, as he gets older his intelligence decreases, as in Fig. 2.4. Suppose his chance of passing to the next social level depends equally on knowledge and intelligence; then the curve for age against chance of increasing social level would be Fig. 2.5.

As increase in age is uniform (one year each turn) but social level is variable, so that keeping your chance at 50 per cent depends on previously being successful (which initially you will be only 50 per cent of the time), the actual curve of chance of success in increasing social level as the game progresses will look more like Fig. 2.6, i.e., initially age and social level cancel each other out in determining the chance of increasing social level, but as time goes on age becomes more and more important. You will see that the design of Mernar Keep involves a number of interrelated curves of this type.

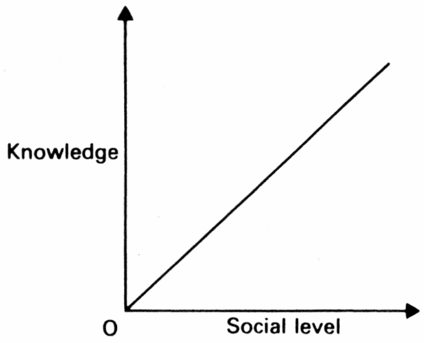


Figure 2.3

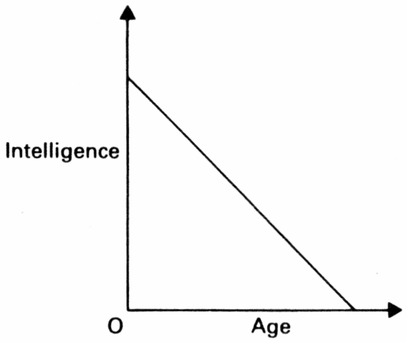


Figure 2.4

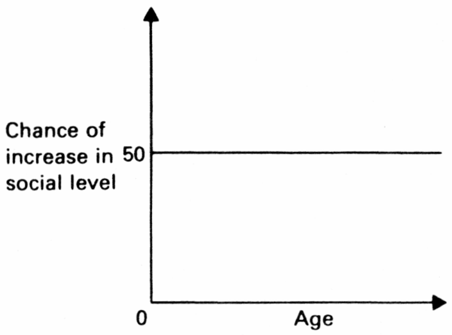


Figure 2.5

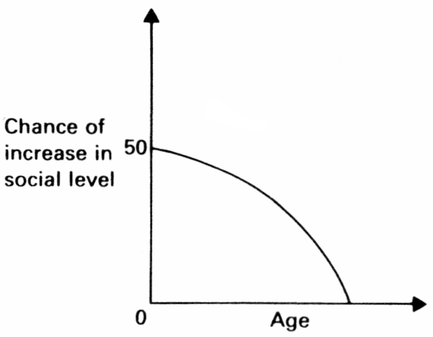


Figure 2.6

Every probability in a game can be given a curve like this, and the chance of winning the game will depend on the curve which summarizes the combination of all the curves. This can involve some very complex mathematics. However, it is not necessary to go so far as to compute all the curves of all the functions in our game, especially if luck (the random factor) plays a large part. What we have to ensure is that we choose functions, formulae, and algorithms which (1) make sense in the world we are creating and (2) never produce 100 per cent certainties. We must work out the consequences of a few sample choices and play-test the most frequently used functions in our game before we finally decide on incorporating them into the game. It is always possible to adjust the formulae later on, but it is much better to have a balanced structure worked out from the beginning so that we do not have to resort to tinkering which could upset the whole game.

You will find some appropriate suggestions for functions at different points in this book as we explore each particular feature.

2.5 Input/output

SOUND

Many micros have no sound facility and others have very poor ones, so you may not wish to incorporate sound into your game. It is certainly less worth while for adventure than for arcade games, so it is by no means essential. However, both the BBC and Electron micros have excellent sound facilities and there are certain uses of sound which can enhance all types of games, so if we decide to use sound we should do so early on in the design process to maximize effective use, as with every other feature of the game.

The main decision we must make initially is whether sound is to be an integral part of the game, giving information which is not available in any other way (such as using musical clues as part of a puzzle), or simply an enhancement containing no essential features (e.g., a tune played at the beginning and end of each game). Naturally sound can be used in both ways in one program. In the case of non-essential sound it can always be added during the final stages of coding, when we know how much memory we have to play with and which sections of the game need such enhancement. However, essential sounds must be built into the design as early as all the other essential information so that we can see how to fit it into the overall structure and be able to develop aural effects parallel to the rest of the program rather than tagging them on superficially.

VISUAL

Unless your game only gives printed hard copy (as in some play-by-mail games) there will always be a visual component. Each turn information will be displayed on the VDU. It may be entirely graphical, in which case it will be an arcade-type game, or it may be entirely textual. With the current generation of microcomputers it seems rather wasteful to have no graphical output, but on the other hand graphics can use a large amount of memory and games like simulations and adventures generally require as much memory as possible for their logic. It is a clever programmer who can build exciting graphics and interesting artificial intelligence into the same game, for example. The Opal Lily in Chapter 9 is entirely textual, but even so it has been difficult to find enough memory for it. The traditional adventure game is similarly entirely textual, but there is an increasing demand for games which use real-time graphics, colour, high resolution, three-dimensional illustration, etc. As memory becomes cheaper such developments will become more practicable.

It is therefore a good idea to make your first game mainly textual as this simplifies matters, but to set aside one or two kilobytes of memory for experimentation with graphical enhancement, such as in titles or score routines.

If, on the other hand, we want an adventure which illustrates every room and every monster, or a game which uses some form of animation, then we will need much more memory, which will limit the size and scope of the program. There are, however, ways of using files on disks or cassettes which allow storage of graphics outside RAM, thereby giving the best of both worlds (but slowing down execution of the program).

For the moment we need to decide:

1. If we wish to use graphics.
2. For what purpose.
3. How much memory we will set aside for this.

Essentially there are three types of graphic display used in games, namely:

1. Decorative display, as in titles, flashing warnings, decorated text, etc.
2. Illustration, as in a picture of the monster the character is about to grapple with or the view seen upon entering a new location.

3. Essential information, where the graphics hold information which is essential to the game and not given in any other way, such as when visual clues are used, or in an arcade-type display.

The first of these is easiest to do and can greatly increase the attractiveness of a game, but being strictly speaking unnecessary is often neglected. The third is the most interesting, but also the most difficult. This book is not about graphics so we will concentrate largely on the second. However, we should always attempt to use a micro to the limits of its resources and one way to do this is by clever mixture of sound, graphics, and text which is not merely illustrative and could not be emulated in a board or table-top game. In Chapter 3 a short game is given which uses no artificial intelligence and requires only minimal intelligence to play, but demonstrates some of the basics of achieving an arcade-like display in a BASIC program.

In planning our display it is a good idea to draw at an early stage rough sketches of the kind of displays we wish to see. While it is unnecessary to go to the lengths of the film-maker's storyboard, on which every shot in the film is drawn before it is photographed, it is a good idea to sketch the key pictures so that we can decide where they will fit into the program, how they will use available screen space, how they can be mixed with text, which graphics mode(s) will be used, and whether our ideas are too ambitious for our skill or machine.

Eventually we will need to plot all the major displays on graph paper or a special plotting sheet, but at the planning stage simple free-hand sketches are sufficient. Bear in mind that an illustration, to be worth while, must be attractive, but a piece of important visual information, such as a clue, can simply be a functional, unaesthetic use of graphics. Remember also that one of the things that makes a player return again and again to a game is its look and 'feel', so paying some attention to how the game presents itself, including its output on the screen, can be very important.

TEXT

Text is the most common form of input/output used in games. We need to consider how text will be used, how it will be presented, how it will be stored, and how it will be processed. The art of adventure design in particular has a great deal to do with variety of text output and versatile analysis of text input. Unfortunately, storing large quantities of text uses large quantities of memory and in BASIC string handling is generally slow, thus slowing down the game.

Therefore some compromises will have to be reached. These are discussed further in Chapters 8 and 9.

We need to consider if all instructions will be included in the program or if they will be described on an accompanying sheet; if commands are to be in normal English or abbreviated words, numbers, or single letters; if some form of data compaction is to be used to store more text; if the description is to be full sentences or abbreviated words; and if it is possible to use commands which can be interpreted in different ways. We also need to decide how to display the text and how to do so in conjunction with any graphic displays. For example, if we want text and graphics together on the screen throughout the game we will need to define text and graphics 'windows' of some kind or even several different windows for different purposes (such as one for a character update and one for a choice of possible actions).

Each of the topics 'screen display', 'data compaction', and 'text processing' deserves a book in its own right, so they cannot be handled thoroughly here. Instead, a number of relevant approaches will be suggested, together with some BASIC coding routines which deal with detailed examples. These can be incorporated at relevant points in your own programs. Essentially, however, the principles to abide by are:

1. Use anything which saves memory.
2. Use anything which speeds execution.
3. Use anything which adds variety to the game.
4. Use anything which makes the program easier to play and more enjoyable for the player.

Because of its fundamental importance it is worth considering text at length in the planning stage. Text processing is often neglected by BASIC programmers despite the fact that many interesting procedures can be carried out. On the one hand we must consider ways of making the game more attractive to the player, in terms of textual variety, correct spelling, easily read displays, etc. On the other hand we must see if we can come up with any original ideas for using text, such as making a pun on the player's name, using jokes in response to player mistakes, displaying text as fragments of parchment found at different places in the game, using cypher routines which change cypher from game to game, generating a series of verbal clues, routines for 'conversation' with encountered monsters, variety in error messages, etc.

2.6 Program structure

When we have sorted out ideas on the story, the game, and the forms of input and output, together with some sketches of what the display will be like, we are ready to begin programming. A game is like any other program but there are some aspects which it is wise to pay attention to and which may modify your normal programming techniques.

Probably most important is the fact that the kinds of game discussed in this book are generally long. To be interesting games must generally be varied and complex as a story/game/puzzle and this means that it will generally be varied and complex as a program. The easiest way to cope with these three related problems of length, variety, and complexity is to adopt a modular structure. Modular programming is a good idea in general because it prevents tangled nets of GOTO statements and encourages BASIC programmers to adopt structured techniques which makes the transition to languages like Pascal and FORTH somewhat easier. Modules enable us to test each section of a program as we develop it and enable us to debug complete programs more easily. Furthermore, modules can be used in different programs with little or no alteration, so that once we have written one game we need never start from scratch with any other. The designs in this book are therefore modular in nature, designed for versatility in use.

What is meant by modular design? Essentially it is the same as dividing your program up into a number of subroutines or procedures, plus a main program which calls each subroutine as it is required, as described above. The supervising program calls all the other procedures which are needed at various points in the program and these procedures may in turn call others. In BBC BASIC the listing of the program therefore consists almost entirely of procedure definition, probably with some definition of functions as well. If you look at the listing of Mernar Keep in Chapter 4 you will see this very well. (In dialects of BASIC which lack procedures the same kind of structure will be created by using GOSUBs and subroutines, but this is usually slower in BBC BASIC and less easy to read in the listing because numbers are used rather than procedure names. However, you will see that the program in Chapter 9 uses both procedures and subroutines for a special reason.)

You should not feel, however, that programs which are not structured are somehow 'wrong' or 'inferior'. The rule in games programming is 'If it works, then it is good programming'. Some people find that structuring imposes too many limits for them to program comfortably; others resent being told that perfectly workable practices are, for some inexplicable reason, bad. If you do use unstructured code you should simply be aware of the pitfalls you may encounter and of the difficulties that other people may have in understanding your work. Of course, in some cases, such as preventing piracy, this might actually be an advantage. The major advantage of structuring is that it imposes a logical order on your thinking which you might otherwise have; this makes you plan ahead and enables you to check your ideas more easily.

In a structured program each module or subroutine has its own particular task—it may print instructions, or calculate combat, or display spaceship movement. That task may be called once or several times per turn. If called only once, and particularly only once per game, there is usually no need to put the module in a subroutine. Instead it can be a clearly marked section of the main program. However, to encourage flexibility and variety it is a good idea for as many as possible of the subroutines to be capable of being called up more than once, depending on conditions. For example, the combat subroutine may be called if the player decides to fight or, later, if the enemy decides to fight, or perhaps even if both wish to avoid fighting but the 'gods' (i.e., a random number) decide otherwise.

If we have done our preliminary designs well it should be clear what modules the program will need and in what order they will be used. If it is not clear then our first programming task is to make it clear by writing down all the ideas we have for the game and linking them together. There are various methods for doing this but I suggest two:

1. The mind map
2. The general algorithm

The mind is a way of generating ideas with links between them. The algorithm is a way of structuring ideas in a logical and systematic way.

THE MIND MAP

Take a blank piece of paper and write in the centre the main idea you have for the program (what it is about, how it will work, what it will look like, what it is called). Now, as rapidly as possible, so you do not have much time to think consciously about it, jot down an idea, phrase, or word which seems to be connected to that main idea. Link the two together with a line. Think of another idea connected with one of the two you now have on the page, write it down, and draw in the link. Now repeat the process with another idea connected to one of these three.

Carry on in this way, thinking up new ideas related to one or more of the ideas you already have on the page, and writing them down, and then drawing all the major links to ideas already down. Do not pause to evaluate any of the ideas. Just jot them down, as fast as you can, and draw in the links. Carry on until you have definitely run out of ideas. You should end up with something like Fig. 2.7. Here the central idea it started with was 'Mernar Keep'.

This is your mind map. It is a plan of the ideas you have about your program or game linked in the ways that make most sense to you at an intuitive level. Now take all the notes, lists, sketches, jottings that you already have on the game and if they have not been incorporated in the mind map then add them and their links at the most logical place(s).

You should find that certain ideas have many branches coming from them, whereas others have only one or two. Those with many branches are the main ideas, which will therefore become the major routines in your program; those with only a few branches will be small modules, used less; and those with only one branch do not need to be separate modules at all but can be included in the larger idea from which they branch. For example, in Fig. 2.7 the topic 'money' has four branches coming from it, so it is quite important—a major

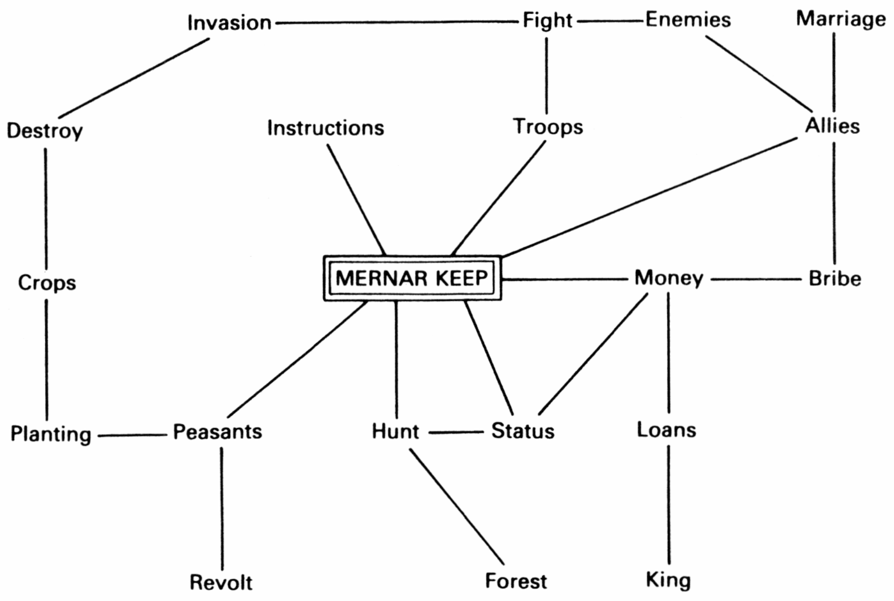


Figure 2.7

routine or set or routines. On the other hand, 'revolt' only has one branch so it is minor. In fact, it will only be called by the 'peasants' routine and therefore there might be no need for a separate routine; the 'revolt' routine could simply be one part of the 'peasants' routine.

THE MAJOR ALGORITHM

A mind map has the advantages of crystallizing all our ideas and clarifying the relationships between them, so we can see what gaps there are and whether the logic is sensible. However, it is not an exact description of a program by any means. The best way to achieve this is probably by constructing a flow chart or algorithm. This is a much more precise model, but is more difficult to construct, particularly if we are still unclear about some aspects of the program. So it is a good idea to construct a mind map first, to generate the ideas, as a way of doing the groundwork, and then turn it into flow charts/algorithms to make the relationships unambiguous. You cannot easily code a program from a mind map, but you can from a well-constructed algorithm.

There are several examples of flow charts and algorithms in this book. Some are quite precise, being quite close to actual BASIC routines; others are very general, leaving some stages implicit or undeveloped. It is this latter kind you should first aim for in planning your game—a broad description of everything that will occur in the program, with some indication of the order in which they will occur. Using the mind map in Fig. 2.7 we might construct an algorithm like the one in Fig. 2.8.

```
(1) PRINT INSTRUCTIONS

(2) REQUEST ACTION

(3) CALCULATE ANY COMBATS

(4) CALCULATE HARVEST

(5) PRINT PLAYER'S CURRENT STATUS

(6) IF THE PLAYER IS NOT DEAD THEN GO

TO (1).
```

Figure 2.8

Having worked out the major algorithm we can then take each block in turn, treat it as a separate module, and draw an algorithm for all the processes involved in it (possibly by brainstorming with another mind map just of that section). Each of these processes should, if necessary, have its own algorithm. We continue in this way until our algorithm begins to read like a program and we are ready to code our module. Assign line numbers to each of the blocks in the most specific drafts and we can then begin to translate the stages of the algorithm into appropriate lines of BASIC code.

Work through the 'instructions' module in Fig. 2.9 as an example. Firstly, we decide that there will be written instructions displayed before the game begins. Then we draw an algorithm like Fig. 2.8 putting the 'instructions' block in the correct place. Now we take a separate sheet of paper and produce an algorithm of the 'instructions' module, perhaps like Fig. 2.9.

```
(1) FLASH UP THE TITLE OF THE GAME TEN

TIMES.

(2) PRINT THE DESCRIPTION OF THE GAME.

(3) DESCRIBE POSSIBLE PLAYER ACTIONS

AND THE AVAILABLE CONTROL KEYS.

(4) PLAYER PRESSES ´S´ TO START THE

GAME.
```

Figure 2.9

Each of the boxes in Fig. 2.9 needs further elaboration. So instruction 1 might be redrafted like Fig. 2.10. This looks very much like a BASIC program. It is only a small step from Fig. 2.10 to a section of code like Fig. 2.11.

```
(1) ADD 1 TO COUNTER

(2) CLEAR SCREEN

(3) PRINT TITLE OF GAME

(4) IF THE COUNTER <> 10 THEN GO TO (1)

(5) CLEAR THE SCREEN

(6) CALL THE NEXT SECTION OF THE

PROGRAM
```

Figure 2.10

```
10 FOR I = 1 TO 10

20 CLS

30 PRINT "Mernar Keep"

40 NEXT I

50 CLS60 PROCmain
```

Figure 2.11

Although this piece of programming is elementary, the same procedure should be used for more complex tasks. In fact, it is more important to use it for complex tasks because without it we may well miss crucial stages of coding when it comes to writing the actual program. Using a series of algorithms like this may seem tedious, but very long programs are prone to many different kinds of bugs. Time we may lose in the planning stage will be amply repaid when we find we have almost no debugging to do.

However, this procedure cannot on its own ensure perfect, bug-free design. Other steps have to be taken. One important thing to do is to ensure that the program is amply documented. Do not throw away any design notes, make sure that all thoughts and changes are written down when thought of so they are not lost, and keep the program full of REM statements. A good idea is to use increments of 10 lines in writing your BASIC programs and place all the REM statements on lines ending with 9. In this way each section of code will have its identifier immediately before it and you will know what lines to look for as you scan through your program during development looking for a key section. Then, when the program is finished, if you wish to delete them (to save memory or to make the listing opaque for a user) it is a simple task to go through and delete all lines ending in 9. An easy way to do this on the BBC and Electron machines is simply to type "Auto 9" and then keep on pressing <Return> until the line numbers generated by the Auto command are greater than the highest line number in your program. What this will do, of course, is delete all lines ending in 9. So make sure before you do it that only REM statements are held on such lines or you may wipe out valuable code.

2.7 Suggested themes for games

FANTASY

1. An underwater world (Atlantis, Mu, Captain Nemo)
2. A world entirely of winter

3. A world in the sky, peopled by winged creatures
4. A world inside a living organism (like The Fantastic Voyage) or a machine (like Tron)
5. The player is to learn the magical skills of a lost race, not through combat but by correctly interpreting a series of magical clues
6. A game based on a series of competing religions or magics, in which similar symbols have differing meanings (hence ambiguous clues)
7. The player designs and plays as a monster of some kind, such as a dragon or vampire
8. A game based on some unusual geographical feature, such as inside a volcano, on a series of floating islands, inside a glacier, among tree tops, or entirely on a cliff face.

SCIENCE FICTION

1. A planet with an unusual shape or topology, not a globe, such as a world which really is flat, or a game based on Larry Niven's Ringworld
2. A game in which the player has the role of a robot
3. A game in which the player is a machine, such as the computer in a starship, the starship itself, or an exploratory vehicle on a new planet
4. The player chooses an alien race to role-play and must behave within the constraints of that race (e.g., a giant insect or an intelligent plant)
5. A game based on time travelling, in which players must pick up clues from different milieux
6. Prevent the mad scientist from conquering the earth.

HISTORICAL

1. A game based on a remote or exotic culture, such as those of China, Japan, Tibet, The Pacific Islands, or on the Incas or Aztecs
2. A game based on a particular historical period or event, such as the rise and fall of Rome, the discovery of America, Alexander's conquest of Asia, the wars between Mongol and Chinese, or the spread of Islam
3. A sporting event or competition used as the theme, such as a round-the-world yacht race, a season of cricket, a Grand Prix
4. Real-world simulations, such as a game on the recording industry, or advertising, or the cinema
5. Spies, espionage, and terrorism
6. An ecological game in which the player must take on a different biological role.

3 PLANNING THE DISPLAY

3.1 Display strategies

A crucial aspect of most games, but particularly computer games, is the physical appearance of the game—what it looks like on the screen. This means that you must devote a great deal of thought to the display of your game, whether it is a graphic or a text game. What will the player actually see? In a graphics game the answer to this question may control almost all other aspects of the game and an adventure game may be the same if, for example, it is primarily a maze to be solved or a real-time game. In the majority of games you cannot let the display be the most important aspect; the structure of the game comes first. However you will need to display both text and graphics, so you must decide how they will be shown and how they will fit together.

The preliminary questions to answer are as follows:

1. Is the display to be purely text, or purely graphics, or a mixture of both?
2. Will there be any real-time interaction and, if so, what will the time interval be for displaying information on the screen?
3. Do we wish to use any of the special features of the machine? For the Electron and BBC this means deciding if we want to use any of the following:

    flashing colours
    high-resolution graphics (what resolution?)
    changing logical colours using GCOL graphics and text windows
    the various different PLOT commands
    user-defined graphics

4. How much information will be shown at a time?

There are three display strategies. The first is successively to add lines to the display so that it is continually scrolling up the screen. This is the easiest method, especially for displaying text, but one of the most untidy and unattractive. It fills the screen with a great deal of information which makes it difficult for the eye to find the exact piece it wants, and most of that information is unwanted at any particular point in the game anyway. Each time it scrolls the whole display moves up in a ragged way and the whole effect can be untidy and uninteresting.

A slightly more convenient and attractive method is to clear the screen at regular intervals. This can be done each time the screen is filled, which saves the need for continual scrolling, but is better after each block of information is shown. This means dividing the information displayed into separate logical classes, each with its own subroutine or procedure, and calling each class when it is required. Each block of information is a 'chunk' of text or illustration to be shown at a particular point, called when it is needed. To do this we need either a highly structured program or we would normally use what is called a menu-driven approach (or both). Roughly speaking, a menu is a set of options displayed on the screen from which the user makes a selection. The user then inputs the selection and the appropriate subroutine or procedure is called, clearing the menu from the screen and displaying the appropriate information for that subroutine. For example, the main menu might read:

Option	Select
Combat	1
Refuel	2
Status	3

If the user wants the combat routine then '1' is typed, and the display changes to that of the combat routine. For example, it might show a graphic display of the view seen by a spaceship's combat computer. Typing '2' might give a different graphics display, let us say a real-time graphic game in which two spaceships shown on screen have to be docked together for refuelling.

It is quite possible that making a selection on one menu could lead to the display of another menu, representing in effect a series of nested subroutines. Thus selecting option 3 above might result in the following display:

Option	Select
Fuel	1
Ammunition	2
Battle damage	3

Selection of one of these could lead to a further menu, and so on until the nesting limit of the microcomputer is reached. A series of such menus is a way of guiding a user through the complex structure of a program to the actual routine required. It is mainly used in business programs and it is unlikely that you will produce a game as complex as such commercial software. However, it is a useful way of relating displays to each other. When the user has found a way to the routine wanted and has carried out the required task by navigating through a series of screens of information (each of which clears the previous screen), the screen will clear again and return to the original menu. This menu is generally known as the 'main menu'. As the technique depends on clearing the screen at the right moment it is a good idea to get into the habit of clearing the screen right at the beginning of each chunk just in case the calling routine itself leaves something inelegant on the screen. The first line of each block of information will therefore be the CLS command.

Using the scrolling method requires no planning or programming at all as the usual method of most microcomputers is to display each line of information at a time and scroll upwards when the screen is full. To clear the screen at appropriate moments, whether using a menu-driven system or not, demands careful programming. For most purposes we need only to divide all output into appropriate blocks, place each block in a subroutine, and make the first command of each subroutine CLS. However, in some cases we might not want to place output in a subroutine or PROCedure, e.g., in the instructions at the start of a game. In this case we must compose each screenful of information so that it is easiest to interpret, placing a CLS command after each successive screen. However, remember here that the screens will clear too quickly for anyone to read, so we must place a delay of some kind at the end of each section.

There are two forms of delay—causing the computer either to perform a process without result for a fixed period or to hold up the program until the user inputs some information. Using the first alternative means that the reader has a predefined period in which to read, which may be too long or too short for some readers, but guarantees that the rest of the program will be carried out. Using the second alternative allows users to read the information in their own time, but involves some action by them to ensure continued operation, so you need to add an instruction explaining what kind of input is required.

The first type of delay can be achieved simply by using a repeated loop, generally a FOR . . . NEXT loop, e.g.,

```
100 FOR I = 1 TO 1000
110 NEXT I
```

The precise interval this achieves will depend on the microcomputer used, so has to be discovered by trial and error. However, the Electron and BBC micros have the INKEY command which waits for a specified time or until a key is pressed. For such micros the best method is therefore to instruct the program to wait for a long time using these commands so that plenty of time is given even for the slowest reader, but faster readers can interrupt at any time simply by pressing a key. Therefore if we decided that it takes 10 seconds to read a particular screen slowly, add a further 5 seconds for good luck, multiply by 100 (because INKEY and INKEY$ wait in hundredths of a second), and use a routine like the following:

```
10 REM FIRST PRINT THE SCREEN
..................
..................
..................
40 PRINT "PRESS ANY KEY TO CONTINUE"
50 INKEY (750)
60 REM NOW PRINT THE NEXT SCREEN
```

In microcomputers which lack such commands, the programmer must choose between the repeated loop (above) or user input. As the input does nothing more than allow the program to continue, it does not matter what that input is, so it is usual to allow any key to be pressed. A line such as:

```
100 A$ = GET$ : IF A$ ="" THEN GOTO 100
```

will be allowable in most dialects of BASIC. The first part of the line looks for the input character while the second part loops back to the first if no character has been input, causing the loop to continue endlessly until a character is typed in.

However, a better (more structured) version in BBC BASIC is:

```
10 REPEAT
20 g=GET
30 UNTIL G<>0
```

This can be adapted so that all keys on the keyboard are disabled (except Break and Escape) while one specific keypress is waited for.

We provide a continuous loop which is always waiting for input from the keyboard and only exits when the right key is pressed, e.g.,

```
10 PRINT “Press space bar to continue”
20 REPEAT
30 g=GET
40 UNTIL g=32
```

This waits until the correct key (in this case the space bar) is pressed. Similarly, we can set up a loop which only exits when one key out of a limited number is pressed. For example, if we wanted to ensure that only ‘Y’ and ‘N’ (for ‘Yes’ and ‘No’) would allow the user to go any further we would use GET$ in the same kind of loop:

```
10 PRINT “Please type Y(es) or N(o)”
20 REPEAT
30 g$=GET$
40 UNTIL INSTR (“YyNn”, g$) <>0
```

Line 40 in this little routine uses INSTR to see if any of the four upper and lower case ‘N’ and ‘Y’ characters have been pressed. If they have not then the loop continues. Obviously any string of characters could be tested for in the same way. It is usual when testing for a range of numbers to use the maximum and minimum values to control the exit, with a line like:

```
40 UNTIL g<3 OR g>7
```

where the desired number must be in the range 3 to 7. We can treat numbers just like any other character and so allow any selection of numerals to be acceptable input, e.g.,

```
40 UNTIL INSTR(“2479”,g$)<>0
```

The third method of display is to clear only those parts of the screen which require updating. The ability to split the screen between text and graphic windows is, of course, one way of doing this, as we will see later on. However, in some cases we might want to preserve, let us say, all the lines of text on the screen except for the scores. Here we need to know where the scores are printed on the screen, to wipe them out with spaces and to PRINT the new scores over the same space. As it is necessary to delete with spaces rather than just overprint the old numbers because the messages may be of different lengths there is a danger that some of the earlier message will be left behind.

To do this we use the TAB function for text and a PLOTting routine for graphics or text treated as graphics. For example, if the score was PRINTed 10 columns along the screen (from the left) and 5 lines down, and the longest score was five characters, then the following routine will wipe it and PRINT the new score:

```
10 DEF PROCscore
20 REM First calculate the new score
30 PRINT TAB (10, 5); STRING$ (5," ")
40 PRINT TAB (10, 5); score
50 ENDPROC
```

The programs in this book have been designed to illustrate some of the different methods of display. Scissors in Chapter 5 uses a continual display of scores and results throughout the game itself but has a screenful of instructions at the start and a short message at the end. Mernar Keep in Chapter 4 uses a menu display to control the major displays but has some scrolling within separate parts of the game. Dilemma in Chapter 5 displays the game board all the time but deletes the pieces and displays messages in a text window. The Opal Lily in Chapter 9 includes its own printing routine because it codes text to save memory, part of which involves 'filling the screen with text' and scrolling off, but clears the screen when a new location is found. This chapter concludes with a small demonstration game illustrating a few of these principles.

3.2 A look at windows

A better method than either continual or periodic screen-clearing is to use screen 'windows'. A window can be thought of as a section of screen defined for a particular type of display. The BBC micro and Electron allow us to define separate text and graphics windows, using separate portions of the VDU screen. In the text window only text appears while in the graphics window only graphics and text treated as graphics appears; these windows may be any rectangular portion of the screen.

Such a capability is, however, non-standard and owners of other micros normally have to write their software routines to create such 'windows'. We use VDU24 to define a graphics window and VDU28 to define a text window. Each command is followed by four coordinates which specify the rectangle to be occupied by that window. Unfortunately, however, we can only have one window of each kind on the screen at the same time. For most purposes one of each is plenty. In a game like Dilemma, for example, which shows only a game board plus some brief instructions, nothing more is needed, but a more complicated game might require a number of different displays simultaneously. For example, a combat adventure might need to display a description of the current location, a graphic picture of that location, the last sentence the user has typed in, the response to that sentence, and an illustration of the monster that has just appeared. Not only must this be arranged on the screen in a way that makes it easy for the player to understand and pleasurable to look at but it has to be done in a way which ensures that no two displays encroach on each other's territories.

However, we can create our own windows by use of the TAB command. With this command we can create windows on any part or parts of the screen that we like. We can write PROCedures which print variables at certain positions on the screen and assign to these variables the actual values or strings we want printed. If the display is to be graphical, the PROCedures will restrict graphic parameters to the predetermined space. This solution is in many ways more elegant and useful than having to put the TAB coordinates in every PRINT statement, because we only have to calculate the necessary coordinates once for each window and then hold them in the PROCedure. But if you prefer you can define text windows so that every kind of PRINT statement must be in a particular place on the screen and must therefore have the appropriate AT or TAB coordinates. This is the strategy used in the program at the end of this chapter.

These basics can be demonstrated easily. Suppose we wanted a list of items the player is currently carrying to be constantly updated in its own window. Presumably in the whole game there will be a large number of things to be carried, but the character will not be able to carry all of them at any one time. This implies that the list will vary from situation to situation. Sometimes there will be no objects and sometimes there will be many. Consequently the techniques of overprinting an exact location will not work, especially as strings tend to be of different lengths. What is needed is a routine which fills as much of the predefined window as is needed with the current text, but also clears the whole of the rest of the window in case the previous text printed in that area took more room.

Let us suppose that no list of items will fill more than five lines of the screen. Therefore we need a five-line window, a routine to clear that window, a routine to put together the actual text from the set of possible strings, and a routine to PRINT the chosen text in the correct window. Let us use the last five lines of the screen in mode 6 (lines 20, 21, 22, 23, and 24) and suppose that the items are listed as items in the array item$(10). From the set of possible variables the program has selected item$(3), item$(5), and item$(8), which are 'a sledgehammer', 'a fir cone', and 'a green and gold necklace' respectively. This list will be preceded by the phrase "You are carrying:".

Firstly, we produce a list of items using the string concatenator (the plus sign!) to turn our separate strings into one string, adding the chosen items to the phrase "You are carrying:". The routine has to know how many items to look for and what they are, so the choosing routines will have compiled a string (L$) made up of the numbers of the selected items in the array item$. In this case L$ will be '358'. The length of the string L$ tells the display routine how many items to look for and add as well as the actual item numbers. This is done by lines 600 to 640 below. Line 650 then clears the selected window by PRINTing five blank lines. Finally, line 660 PRINTs the new string at the correct position:

```
600 REM TO PRINT A COLLECTION OF STRINGS AT
    A PARTICULAR POSITION
610 p$="You are carrying: "
620 FOR i=1 TO LEN(L$)
630 p$= p$ + " " + item$(VAL(MID$(L$,i,1)))
640 NEXT i
650 PRINT TAB(0,20);'''''
660 REPEAT
670 space=INSTR(p$," ")
680 PRINT TAB(0,20);LEFT$(p$,space-1)
690 p$=RIGHT$(p$,LEN(p$)-space)
700 UNTIL p$=""
```

The same procedure can be used whatever the number of lines of screen or the number of items in the list. Simply change the number of apostrophes in line 650 and the TAB numbers. From this example you should be able to see how text windows can be defined almost anywhere on the screen. Every separate PRINT position is itself a text window, so the maximum number of windows would be the number of these positions (which depends on the mode you are using), though you will seldom want more than two or three. The key point is to decide as early as possible how many windows you require and of what size. It makes sense to write the PRINTing subroutines before you have decided on all the text you are to display—but not until after you have classified the types of text you will show. It also makes sense to draw a rough sketch of the windows making a full screen display in the planning stage so you can get an idea of how cluttered or organized the screen will appear to a user.

The major advantages of using text windows are three. Firstly, it produces a clear and attractive display, easy on the eye, well organized and easy to understand. Secondly, it simplifies the process of deciding what to print, where, and when. Thirdly, it means that a large amount of information can be displayed on the screen at a time without chaos. Several classes of information can be displayed simultaneously without the need for scrolling, menus, or constant screen clearing. Of course you can also use one window for several purposes, as if it was a miniscreen. If you wish to do this, and the window-cleaning routine is complex, it is often better to have this as a PROCedure separate from the PRINTing routine, but called by it. In this way several different types of text can use the same cleaning routine and the same window without the need to duplicate code. All that is needed is that the window-cleaning routine be given the parameters of the screen area it is to clear. Thus one PROCedure can be used to PRINT blank strings over any predetermined area of the screen. In a similar way another PROCedure can be given parameters for overprinting any area of the screen with the current graphic background colour, in this way 'cleaning' graphics windows.

3.3 A basic graphic game

This book is not primarily concerned with graphic arcade-type games or indeed with graphics. The BBC and Electron micros have such a bewildering and versatile array of graphic facilities that justice could not be done to them here. However, some of the principles of game design can be shown by creating an elementary graphic game, so that is what this section does. Other chapters deal with aspects of the games such as structure and various aspects of intelligence. In this section we will concentrate on the simpler principles of producing an elementary entertaining graphic game to illustrate some of the arcade principles, some of the BBC/Electron's features, and some of the more general considerations to bear in mind when designing games of any kind and, in particular, aspects of display.

Speed is one common feature of most graphic games but even in BBC BASIC it is difficult to achieve a satisfactorily breath-taking invaders-type game. Arcade games have their attractions and there is no doubt that one of the major reasons for the growth of the home computer industry has been the popularity of fast-action screen games. It is not possible in BASIC to write very fast arcade-type games—the language is just too slow, because it is an interpreted language. If you have a compiler or understand machine code, you will not have this problem, but the majority of us have to make do with the slowness of BASIC. Fortunately we have a useful assembler built into Acorn's machines so it is easier to begin to get to grips with machine code than on some other machines. Hopefully by the time you have finished this book, particularly when you have read the final chapter, you will have enough confidence in handling bits and bytes to begin experimenting in Assembler, but you do need quite a substantial knowledge to create fast-moving games that are at all satisfactory.

However, speed is not everything. The growing popularity of adventures shows this. Some designers have attempted to put a little arcade action into adventures but with relatively little success. Broadly speaking, when you design a game you will thus be making a choice between a strategy/problem-solving game, of which the adventure is the main type, or a game of reaction time and speed with moving graphics. There are strategic elements in arcade games, however. No player of arcade games survives by reaction time alone. A strategy is needed which takes into account the likely behaviour of the alien blobs, the rewards available for destroying each particular type of alien, the time left for play, and so forth. A number of factors have to be mentally assessed at the same time as the player's fingers are flashing over the keyboard.

The problem can be made harder for the player if some of the aliens or the pursuing blobs have a degree of intelligence. This could be given in several ways. One would be to ensure that the enemy did not just become stronger or faster or more numerous, with increases in the player's score, but that they also adapted their behaviour more to the tactics of the player. For example, the lowest skill level might have the ghosts in a pacman game moving about randomly. The next level up might be written so that, on average, every fourth move by a ghost was in the general direction of the player. At the next level it would be one move in three, at the next one in two, and finally every move that the ghosts made would be following the player.

Another tactic could be to program aliens to respond to the player's strategy for shooting at them. Suppose the player always hid behind one particular defence or always attempted to stay roughly in the centre of the screen. It is relatively easy to monitor this sort of thing by simply incrementing a variable each time the player is hiding in a particular position. The aliens could respond to this 'knowledge' in a number of ways, perhaps by concentrating their attacks on this area, or by ensuring that the high-scoring targets avoided that area, or even by launching 'intelligent' missiles which always homed in on the area most commonly frequented by the player. There is no difference between this kind of sensitivity to a player's preferred actions and that discussed in the next chapter in the general context of 'strategic' or 'abstract' games. As yet the sheer thrill of speed, sound, and colour has created most of the attraction in this type of game but it is only a matter of time before the aliens not only have faster reactions than the player but also understand more about how to play the game.

However, we will not design such a complex game for our first effort. A simple type of arcade game will serve to illustrate the need to develop strategy. One which is particularly easy to design is a treasure-gathering routine. We remove some of the difficulties of arcade programming by keeping the treasure stationary and only moving the player's representative. The player then simply has to determine the best route to wander around the screen collecting his or her ill-gotten gains.

There are only two real problems in arcade games—moving objects and collision detection. In brief, moving an object involves printing a graphic block in one position, calculating a new position, printing it in the new position, and deleting it from the old position. If this happens fast enough, it looks like movement. Collision detection involves adding a number of tests within the movement loop to see if the new position is already occupied and if so what the occupier is. The results of the two objects meeting are then calculated and all the relevant variables, such as the score, are updated.

It is easy to see why this slows a game down. If we have 40 invaders, one gun, five missiles, and 20 bombs on the screen at the same time, then each 'turn' in the game involves deleting 66 objects, calculating their next positions, checking for contacts at those positions, calculating the effects of any contacts and displaying the results, and printing 66 objects at their new positions. For all this to happen and still look natural it must literally take place in the blink of an eye. For BASIC it is more like 40 winks.

So we will just look at a simple version in which only one object moves (a little man representing the player) and only one position needs testing for collisions each turn (the position the man will next move to). The idea behind the game is simple. The player must move his piece around gathering food and drink but avoiding exercise. Each time food is encountered he gains bonus survival time; each time he has to exercise (by colliding with a barbell) he loses time. The more you collect the longer you last but as time goes by the remaining food is further and further apart so the distances are greater and energy needs conserving. In addition, each time food or drink is taken a barbell replaces it, adding to the maze of hazards.

This introduces a limited element of strategy—given a restricted amount of time, what is the best route to pick up as much as possible of the high-scoring food while avoiding as far as possible the need to do any exercise? Is it better to collect food first because there is more of it or to gather the delicacies from the edge of the screen before time runs out? Points are scored for each item of food eaten or bottle drunk, so the player wants to get as many resources as possible before time runs out. The more you consume, the greater your score and therefore the longer you last. The competitive element, as with all such games, comes in receiving a final score which the player will constantly attempt to improve. The game also has a number of different levels of difficulty, which basically means that the time limit is shorter at the higher levels.

The program works quite simply and is given in Fig. 3.1.

```
 10 MODE 7
 20 MODE 2
 30 CLS
 40 PROCinit
 50 PROCtitle
 60 PROCstart
 70 COLOUR 128
 80 PROCmap
 90 PROCmove
100 PROCfinish
110 END
120
130
140 DEF PROCmove
150 xm = 640
160 ym = 640
170 g = 1
179 REM REDEFINE CURSOR KEYS
180 *FX4,1
190 REPEAT
199   REM PRINT THE PLAYER´S PIECE
200   PROCman(xm,ym,2)
209   REM PRINT THE GREEN MAN
210   PROCman(x,y,7)
220   FOR I = 1 TO 200
230     NEXT
240   h = INKEY(1)
250   IF h<>-1 THEN g = h
259   REM UNPRINT THE PLAYER´S PIECE
260   PROCman(x,y,7)
269   REM UNPRINT THE GREEN MAN
270   PROCman(xm,ym,2)
280   j = RND(3)-2
290   k = RND(3)-2
300   xm = xm+(j*64)
310   ym = ym+(k*32)
320   h = INKEY(1)
```

Figure 3.1 'Pantry Panic' (*continues*)

```
330   IF h<>-1 THEN g = h
340   IF g = 136 THEN x = x-64
350   IF g = 137 THEN x = x+64
360   IF g = 138 y = y-32
370   IF g = 139 y = y+32
379   REM STOP THE PLAYER WANDERING OFF THE SCREEN
380   IF y>992 THEN y = 992
390   IF y<96 THEN y = 96
400   IF x<64 THEN x = 64
410   IF x>1152 THEN x = 1152
419   REM STOP THE GREEN MAN WANDERING OFF THE SCREEN
420   IF ym>992 THEN ym = 992
430   IF ym<96 THEN ym = 96
440   IF xm<64 THEN xm = 64
450   IF xm>1152 THEN xm = 1152
460   PROCcalc
470   PROCupdate
480   PROCmc
490   UNTIL TIME >((800+score)*diff)
499  REM RETURN CURSOR KEYS TO NORMAL USE
500 *FX4,0
510  ENDPROC
520
530
540 DEF PROCinit
550 DIM board%(20,32)
560 score = 0
570 y = 128
580 x = 128
590 score=0
600 VDU 23,226,&1C1C;&3E08;&0808;&2214;
610 VDU 23,227,&787F;&7C78;&CCEC;&EECC;
620 VDU 23,228,&0000;&0018;&0000;&0000;
630 VDU 23,229,&0D18;&030F;&0000;&0000;
640 VDU 23,231,&4200;&4242;&4242;&0042;
650 VDU 23,232,&4200;&FF42;&42FF;&0042;
660 VDU 23,233,&7EE7;&0008;&0000;&0000;
670 VDU 23,234,&0800;&7F3E;&7F7F;&1C3E;
680 VDU 23,235,&1818;&3C3C;&3C3C;&3C3C;
690 VDU 23,236,&0000;&1800;&0018;&0000;
700 VDU 23,240,&FFFF;&FFFF;&FFFF;&FFFF;
710 ENDPROC
720
730
740 DEF PROCtitle
750 COLOUR 135
760 CLS
770 a$ = "PANTRY PANIC"
780 FOR I = 1 TO LEN(a$)
790   REPEAT
799     REM CHOOSE FOREGROUND AND BACKGROUND
800     R = RND(7)+8
810     S = RND(7)+8
820     UNTIL S<>R
```

Figure 3.1. *(continues)*

```
 830   COLOUR R
 840   COLOUR 128+S
 850   PRINT TAB(I+2,I+6); MID$(a$,I,1)
 860   NEXT
 869  REM DELAY
 870  TIME = 0
 880  REPEAT
 890    UNTIL TIME = 400
 900 ENDPROC
 910
 920
 930 DEF PROCapple(x%,y%)
 940 VDU 5
 950 GCOL 3,1
 960 MOVE x%,y%
 970 PRINT CHR$(234)
 980 GCOL 3,2
 990 MOVE x%,y%
1000 PRINT CHR$(233)
1010 VDU 4
1020 ENDPROC
1030
1040
1050 DEF PROCbottle(x%,y%)
1060 VDU 5
1070 GCOL 0,6
1080 MOVE x%,y%
1090 PRINT CHR$(235)
1100 GCOL 0,1
1110 MOVE x%,y%
1120 PRINT CHR$(236)
1130 VDU 4
1140 ENDPROC
1150
1160
1170 DEF PROCbarbell(x%,y%)
1180 VDU 5
1190  GCOL 0,0
1200 MOVE x%,y%
1210 PRINT CHR$(240)
1220 MOVE x%,y%
1230 GCOL 3,7
1240 PRINT CHR$(232)
1250 GCOL 3,6
1260 MOVE x%,y%
1270 PRINT CHR$(231)
1280 VDU 4
1290  ENDPROC
1300
1310
1320 DEF PROCmap
1330 CLS
1340 FOR I = 1 TO 20
1349    REM PLACE BARBELLS
```

```
1350   REPEAT
1360     R = RND(19)
1370     S = RND(30)+1
1380     UNTIL board%(R,S) = 0
         AND board%(R+1,S) = 0
         AND board(R,S+1) = 0
1390   board%(R,S) = 1
1400   PROCbarbell(R*64,S*32)
1410   NEXT
1420 FOR I = 1 TO 15
1429   REM PLACE APPLES
1430   REPEAT
1440     R = RND(19)
1450     S = RND(30)+1
1460     UNTIL board%(R,S) = 0
1470    board%(R,S) = 2
1480    PROCapple(R*64,S*32)
1490   NEXT
1500 FOR I = 1 TO 8
1509   REM PLACE BOTTLES
1510   REPEAT
1520     R = RND(19)
1530     S = RND(30)+1
1540     UNTIL board%(R,S) = 0
1550   board%(R,S) = 3
1560   PROCbottle(R*64,S*32)
1570   NEXT
1580 PROCborder
1590 ENDPROC
1600
1610
1619 REM PRINTS THE MAN
1620 DEF PROCman(x%,y%,z%)
1630 VDU 5
1640 GCOL 3,z%
1650 MOVE x%,y%
1660 PRINT CHR$(226)
1670 VDU 4
1680 ENDPROC
1690
1700
1710 DEF PROCborder
1720 VDU 4
1730 FOR I = 1 TO 18 STEP 2
1740   COLOUR 140
1750   COLOUR 11
1760   PRINT TAB(I,0); CHR$62;
1770   PRINT TAB(I,30); CHR$62;
1780   COLOUR 139
1790   COLOUR 12
1800   PRINT TAB(I+1,0); CHR$62;
1810   PRINT TAB(I+1,30); CHR$62
1820   NEXT
1830 FOR I = 0 TO 29 STEP 2
```

Figure 3.1. (*continues*)

```
1840   COLOUR 140
1850   COLOUR 11
1860   PRINT TAB(0,I); CHR$62;
1870   PRINT TAB(19,I); CHR$62;
1880   COLOUR 139
1890   COLOUR 12
1900   PRINT TAB(0,I+1); CHR$62;
1910   PRINT TAB(19,I+1); CHR$62;
1920   NEXT
1930 PRINT TAB(0,30); CHR$62;
1940 PRINT TAB(19,30); CHR$62;
1950 ENDPROC
1960
1970
1979 REM UPDATE CALCULATIONS
1980 DEF PROCcalc
1990  a% = x DIV 64
2000 b% = y DIV 32
2010 IF board%(a%,b%) = 0 THEN ENDPROC
2020 IF board%(a%,b%) = 1 THEN PROCtime
2030 IF board%(a%,b%) = 3 THEN PROCbonus(20)
2040 IF board%(a%,b%) = 2 THEN PROCbonus(10)
2050 ENDPROC
2060
2070
2080 DEF PROCtime
2090 FOR I = 1 TO 50
2100   COLOUR I
2110   PRINT TAB(1,30);"****EXERCISING****"
2120 NEXT
2130 ENDPROC
2140
2150
2160  DEF PROCbonus(c%)
2170 score = score+c%
2180  board%(x DIV 64,y DIV 32) = 1
2189  REM TURN APPLE OR BOTTLE INTO BARBELL
2190  PROCbarbell(x,y)
2200 ENDPROC
2210
2220
2229  REM PRINT UPDATED INFO
2230 DEF PROCupdate
2240 a% = x DIV 64
2250 b% = y DIV 32
2260 COLOUR 129
2270 COLOUR 7
2280  PRINT TAB(12,0);"   "
2290  PRINT TAB(5,0);"SCORE ";score
2300  PRINT TAB(12,30);"   "
2310 max = (800+score)*diff
2320  PRINT TAB(1,30);"TIME "; TIME ;" MAX ";max
2330 ENDPROC
2340
```

```
2350
2360 DEF PROCstart
2370  REPEAT
2380    COLOUR 135
2390    COLOUR 0
2400   CLS
2410   PRINT ´´´´
2420   PRINT "Difficulty level"´´"        (1-9)?"
2430    PRINT
2440   PRINT ´"(1 is hard, 9 easy)"
2450   g$ = GET$
2460   diff = VAL(g$)
2470   UNTIL diff>0 AND diff<10
2480  ENDPROC
2490
2500
2509 REM TURN FOOD INTO BARBELLS IF EATEN BY GREEN MAN
2510 DEF PROCmc
2520 a% = xm DIV 64
2530 b% = ym DIV 32
2540 IF board%(a%,b%)<>0 THEN PROCbarbell(xm,ym)
2550 ENDPROC
2560
2570
2580 DEFPROCfinish
2590 PRINTTAB(1,15);"YOU´RE EXHAUSTED"
2600 ENDPROC
```

Figure 3.1 'Pantry Panic'

3.4 Pantry Panic

The program is very simple. The player controls a white man who can be moved around the screen by using the cursor keys. If the man collides with an apple or a bottle he gains points and the food or drink is converted to a barbell. The points scored are used to compute the maximum time limit for the game. The more Igor scores the greater the limit (i.e., the longer the game will play). The length of time is also determined by the difficulty level set at the beginning by the player.

If the white man hits a barbell the game pauses while exercise is taken. Everything stops except for the TIME counter. The BBC and Electron micros have a built in real-time clock monitored by the variable TIME and it is this which is used to check whether the game has exceeded the allowed limit. To make things a little harder a green man is also wandering around doing the same thing. He is moved randomly but each time he finds food or drink a new barbell is added to the screen and a potential scoring item is removed. However, the green man does not have to perform any exercise. (If the two men collide nothing happens other than a temporary change in colour.)

The strategy is thus relatively limited. The player must move the white man as quickly as possible to as much of the food and drink as possible before the green man gets there, and without hitting a barbell. The more food gathered the more difficult the task becomes but the longer the interval available. Eventually, however, there is no food or time left and an end mesage is printed along with the final score.

Even with only a limited game of this kind both men flicker rather too much because they are constantly being printed and deleted. To reduce this a pause has been added at lines 220 to 230 which momentarily halts the game while the white man is displayed.

Both the structure and the display of this game are simple. Let us first look at its presentation and then how it works. There are three screens in the program. Firstly, there is a title screen called by PROCtitle. It uses two very simple devices to make a slightly unusual title. The idea is that it will catch the eye and make the game seem attractive. Line 750 selects a background colour. Every number above 127 after the colour statement is a logical colour for background, so 128 would be a black background and 135 is logical colour 7 plus 128. In mode 2 this is white but you could change it to any background. Line 770 declares the string a$ as the title. Obviously by substituting any string here the same routine will produce a decorative version of that string. This is one of those advantages of modular programming that I have mentioned. With just one alteration the same routine could be used for any title page you wish in any program.

A FOR . . . NEXT loop prints each character in a$ one letter at a time, using the loop's variable I to increment both the horizontal and the vertical TAB positions and thus printing the string diagonally on the screen. Each time a letter is printed the variables R and S are randomly set to numbers in the logical colour range, thereby choosing random background and foreground colours for each character printed. Finally, a delay using TIME holds the display on the screen for a few seconds. TIME can be used to control a delay loop here because we have not yet begun the main program, but as it is used as a controlling variable in the main body another delay mechanism has to be used—hence the FOR . . . NEXT loop at line 220.

The second screen is not a title page so does not need to be very attractive. It is simply a message requesting information to be input, namely, the difficulty level of the game. It is called as PROCstart. All this routine does is print the message a little way down the screen, with a little spacing to make it easier to read. As the number requested is simply in the range 1 to 9 GET$ can be used to gather the value and save the user the added work of having to press the <Return> key. Remember, however, that GET$ has to be placed in a loop (lines 2370 to 2470) to ensure that only the allowed input is taken in.

The third screen is the main screen on which the game is played. It has to hold a fair amount of information so in the design stage it has to be thought about more carefully than the others. We need to display the current score, the time elapsed, and the maximum time allowed, as well as the graphics of the game, and we need to do it in as attractive a way as possible.

The chosen method is to display a border around the whole screen and print the updated values periodically within that border. We can make the border attractive by using alternating flashing colours and by overprinting certain portions of that border using the TAB command and printing in a contrasting colour. Flashing blue and yellow is used for the border and black on red for the information. Each time the main loop (called PROCmove) is cycled through the values are updated and the screen windows in the border for score, time, and max are altered. Occasionally, however, we need one further piece of information to tell the player when the white man has run into a barbell and is thus exercising. As the other information is unimportant at such a point (because the player can do nothing about it except wait) we can overprint the existing windows with this message; thus the bottom border is used to flash the message "exercising".

The rest of the display on the main screen is the game itself. As suggested above this consists of four phases. Firstly, the screen and all variables are set up in PROCinit; then the player moves his piece around the screen; with each movement a check is made for possible consequences; and finally when time runs out a brief ending message is displayed.

PROCinit dimensions an integer array called board%. (An integer array can only hold whole numbers and uses only four bytes for each number, whereas a normal numeric array holds what are called 'real' numbers which can include decimal points. The integer array is indicated by the % sign.) This array will hold all the information we wish to know about the screen. It is therefore a kind of 'map' of the screen. You will find that most microcomputers are described as having 'memory mapped displays' or some similar phrase. This simply means that the micro holds a complete map of everything on the screen at any one time. Consequently our array is, strictly speaking, unnecessary because if we knew how to read the

Electron's own screen map we could use the built-in information instead of holding our own in a separate array. However, finding and using the required information is not an easy task, especially in mode 2, and we only need a crude map, not a map of every single dot on the screen; thus the array is the simplest and most appropriate method.

Having defined the graphics characters in lines 600 to 700 PROCmap randomly fills board% with the numbers 1, 2, and 3 (lines 1350 to 1570). Each number represents a particular object: 1 is the barbell, 2 is the apple, and 3 is the bottle. Each time a number is assigned to one of the array elements the corresponding position on the screen is printed with the appropriate graphics. What do I mean by the corresponding position? You will see that board% is an array of 20 elements by 32, which is the same as the number of character positions on a mode 2 screen. Therefore any element in the array can be regarded as a 'map' of a particular character position on the screen. Array element 1,1 would be the same as TAB position 1,1. If we want to convert this into graphic coordinates then we multiply the horizontal coordinate by 64 and the vertical coordinate by 32. So board% (2,3) holds the number corresponding to graphic screen position 128, 96.

The three drawing routines PROCbarbell, PROCapple, and PROCbottle use this information to draw the appropriate graphics on the screen in the correct positions. One feature of these graphics is that each character has more than one colour. This is done quite simply by overprinting the same character position twice with two different defined graphics. Thus the red part of the apple is CHR$234 and the green part is CHR$233. If two characters are overprinted as text (e.g., by using TAB), such as in our message display routines, then the second character completely overwrites the first, printing both foreground and background. If, however, the graphics and text cursors are tied together using VDU5 then text can be printed using graphics commands, in which case only the foregrounds are printed. This is what is done here. Unfortunately the process is rather slow so the men figures do not have this feature as it would make the flicker even more noticeable.

Having filled the map and thus the screen with objects the main routine, PROCmove, looks for input from the player and moves the figure accordingly while also moving the green figure. The same process is used for both of these figures. A vertical screen coordinate is calculated, a horizontal screen coordinate is calculated, and the results together with the colour (white or green) are passed to PROCman which prints the figure at those coordinates in that colour using GCOL3. The loop is then finished and the figure is printed again at the same coordinates which, of course, wipes it out. New coordinates are calculated and the process repeated.

The only difference between the two processes is in the way that the coordinates are worked out. In the case of the green man random values are added to the base values of xm and ym. In the case of the white man INKEY is used twice on each loop to see if any key has been pressed by the player. If it has and it is one of the cursor keys then the screen coordinates are updated appropriately (lines 340 to 370). In order to use the cursor keys in this way, i.e., as normal keys giving an ASCII code, they have to be disabled using *FX4,1. (If using this feature you should also make sure that *FX4,0 is used to enable them again when the program is over or you will find the editing features of BBC BASIC severely affected). If no key press has been detected since the last execution of the loop then the coordinates are increased by the same amount as on the previous cycle and the figure will constantly move in the same direction until a key is pressed or the border is reached.

Each cycle of the loop calls PROCcalc, which looks to see if the current position of the player's piece coincides with the position of another object by looking at the corresponding element in board% and seeing if it is anything other than 0. If it is 0 then PROCcalc ends. If it is 1 then the game pauses and the exercising message is printed. If it is 2 or 3 then the score is increased and a barbell is printed in the new position. An important point to note is that not only must the barbell be printed on the screen but that element of board% must be changed to 1 because it is this screen map which is being tested and interpreted, not the screen itself. There is a function for testing the colour of a particular point on the screen in BBC BASIC. POINT will return the logical colour of a particular position on the screen. However, as we have used multicoloured graphics in this game, the detection is not that easy to do using this function.

PROCupdate prints the new values on the screen and PROCmc carries out similar checks on the current position of the green man to see if any food has been found and any reprinting needs doing. Finally, when the condition in line 490 is met the main procedure ends and the final message is printed in PROCfinish.

One final note on this and the other programs in this book concerns the format for presentation. A number of blank lines have been included to separate procedures and to aid reading. These, of course, do not need to be typed in to make the program RUN, as with REM statements. If you want to include such blank lines in your own programs simply type a line number, a blank space, and <Return>. The micro does not treat a space as an error but it does ignore it, so such a line does nothing else but waste memory.

In addition a special format has been used for listing in order to make the listings readable. You should not try to copy the shape of the listing onto your screen. Only press the <Return> key at the end of each line, i.e., when it is time for a new line number. For example, line 1380 has been split into three lines of listing but it is still only one line of program.

4 TESTING INTELLIGENCE: SIMULATIONS

4.1 The intelligent player

Having looked at some principles of program design and a basic graphic game let us start our work on a more extensive program, though without yet considering artificial intelligence as such. Instead we will work on a game that tests the player's intelligence without being particularly intelligent itself. This will be useful practice both in applying some of the lessons of previous chapters and in examining what we mean by 'intelligent' in the context of playing a computer game.

There is a growing interest in using computers in the home as a way of gaining insight into various complex aspects of reality which the average human being never has a chance of experiencing. These are programs which model or simulate some particular real situation, such as solving the problems a football manager must face in getting his team to the top of the league, or governing England for 15 years, or flying a Jumbo Jet. Whether you call these games or not depends on your attitude to such programs as well as the nature of the programs themselves. The more complete they are (and thus the more complex) the closer they correspond to the real situation and the greater the degree of 'education' such programs are likely to involve. It is impossible to play a good economic game well without learning something about the nature of economics. Such games and simulations have long been used in colleges and schools as a pleasant way of introducing students to various kinds of learning that they could not possibly encounter in reality. Few people get a chance to fly a Jumbo Jet in real life and those that do generally only have one opportunity for crashing it.

Some models and simulations are used simply because it is not possible to know the actual phenomenon at first hand. Models of subatomic physics portray a world which no-one will ever be able to see at first hand. Models of history try to reconstruct events and ways of life which no longer exist. The keys to such programs are thus careful design, a good knowledge of the subject, and the ability to balance all the various relevant factors of the real world without introducing too many extraneous and confusing details. There is no point in using a simulation as a means of learning if the user is totally confused by the experience.

As we are mainly concerned with games for entertainment rather than education we do not need to worry too much about producing a model which is an exact replication of reality. However, we do need to produce a program which is well balanced and contains a number of different factors. One of the things we recognize in intelligent behaviour is the ability to cope with a number of different and interrelated variables. If we want a game which is not just a test of knowledge and is not extremely simple to play well, we could try to produce such behaviour. The player will have to think about the implications of his or her actions in several areas simultaneously and try to achieve a balance between different kinds of strategy.

The most popular games which operate in this way are economic, political, and war games. They deal with very intricate worlds in which very few people have ever managed to succeed for long because of the large number of factors involved. In this chapter we will devise a game which incorporates elements of the political, the economic, and the strategic—though in a relatively simple way. What we are mainly concerned with is producing a nicely balanced test of a player's intelligence.

The game could be called a historical simulation. It aims to model some of the problems facing a mediaeval or feudal baron. In this case the baron is the player, lord of Mernar Keep. He is one of three barons on an island with whom he is constantly competing for land, labour, wealth, and social standing. In particular, he has to keep in the good books of the King, his feudal overlord, as do the barons of Ardan Keep and Hale Keep, so none of them can be too ruthless in their behaviour. However, they cannot be too weak either or their land may be stolen, their peasants may revolt, or their men-at-arms may starve.

The game is carried out in seasons, with four seasons forming a year. Each season Lord Mernar may take any of a number of actions, and may be compelled to take others whether he likes it or not. He should decide on how much grain is planted and how much land, if any, is to be reclaimed by burning away forest, remembering always that if too many people are at work in the fields when another baron attacks the results may be fatal, and also that he must conserve enough forested land to please the king and to give himself a good hunt. Hunting is one way of gaining or losing prestige in the feudal world and can also provide a useful bonus to the larder in lean times.

Prestige is the main concern of Lord Mernar. He wants to build as good a reputation as possible to gain the royal favour. If his prestige is high then he may find it easier to bend the law, and if it becomes high enough he may find himself made Prince of the whole island with Ardan and Hale banished to the wilds of Milton Keynes. Prestige may be gained (or lost) in many ways, but is primarily a function of the amount of treasure in the coffers and the number of territories held. It is also affected by alliances (made through giving land or money or by marriage), by military success, by satisfactory handling of the peasants (just the right mixture of ruthlessness and ruth), by pleasing the King, by success in the hunt, and by having a well-kept castle.

This general sketch of the concerns of such a lord can be our starting point for setting up the game, and we could use this as the starting point for a mind map, as we have already seen in Chapter 2. Although there is quite a large scope for graphics we are more concerned with the internal structure of the game so we will content ourselves with a symbolic map and concentrate on making an interesting textual game. It is a good idea, however, to get a sketch of the screen display and a list of the likely major routines as an initial guideline, so the first task for this game would be to provide these. The display for Mernar Keep looks like Fig. 4.1 and the list of major routines is Fig. 4.2.

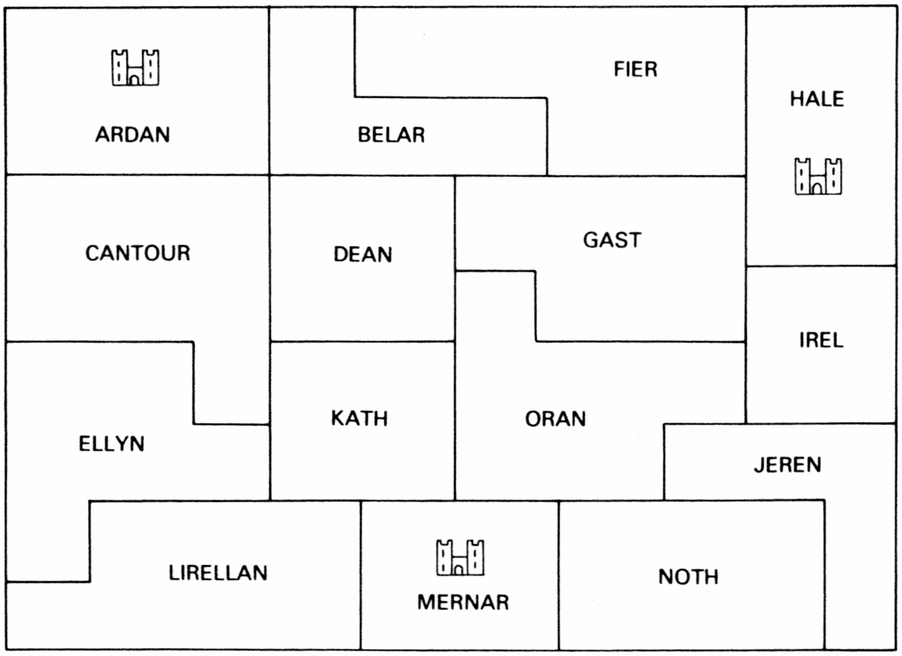


Figure 4.1 Map display for Mernar Keep.

10-999	Set up and main loop
1000-1999	Actions involving enemy lords
2000-2999	Map printing
3000-3699	Mernar´s selection menu
3700-4999	General purpose and end of move routines
5000-7999	Mernar´s main routines
8000-8359	End routine
8360-8999	Function definitions
9000-9009	General Data
9101-9999	Data for lands

Figure 4.2 Block diagram of Mernar Keep.

You will see that almost all the routines are called as PROCedures. This is by far the most satisfactory way of building a BASIC program on the Electron or BBC micros. In using PROCedures we are using some at least of the principles of structured programming, an approach that makes understanding and debugging programs so much easier. You will see that the listing for Mernar Keep is surprisingly readable. This is mainly due to its structured nature. All the major routines are isolated in their own separate block of code as described in Fig.4.2. Each block has been isolated in the listing by prefacing it with two blank lines. These are not REM statements but are simply empty lines with one space typed on. As BBC BASIC treats a line with a space on it as a line of program, it is remembered when typed in; but as spaces are ignored when a program is run (except in a few special cases) they are not treated as errors during execution. Naturally, if you put anything else on these lines other than a space it will be treated in the program as an error unless the line includes an acceptable BASIC keyword.

I have also included a large number of REM statements to explain all the various stages of the program—a large number of spaces which are, strictly speaking, unnecessary—and I have used long variable names in many cases to make it more readable. Finally, the program has been listed using the LISTO7 option.

However, if you type in this program exactly as listed you will find that it does not run. Almost certainly you will get the error "No Room", meaning you have run out of memory. Not only this, but every space and every REM statement and every character in a variable name over and above the first character will slow the program down in running. Consequently, to make this program RUN you should remove all the REM statements and all the lines which only contain spaces. You may wish to substitute short variable names for the longer ones I have used. The main variables are listed in Fig. 4.3. If you are trying to save enough memory to add your own routines to the game you will want to remove all the superfluous spaces and create as many multistatement lines as possible. (A multistatement line is a line of program which consists of several statements, separated by colons.) You may like to keep two versions of the program—one fully documented, structured, and spaced to be used for reference and as a guide when developing similar games and the other streamlined so that it actually runs and does so in as efficient a way as possible. You will find as you learn more about programming that one of its unfortunate laws seems to be that there is an inverse relationship between ease of use of a program from a human point of view and ease of use, or efficiency, from the computer's point of view. People like lots of redundancy (extra information such as REM comments). Machines like the barest minimum of information necessary for carrying out the task.

Also, to save memory (and, to some extent, typing) I have not included any instructions in the game itself. The program is menu driven, as described in Chapter 3, so once you understand its basis you should not need any additional information to play it. When it begins you are told the starting season and the current weather. You are then shown the main menu which consists of all the options Lord Mernar can carry out in a season. Some of them can only be attempted once a season; others can be done as many times as you like. One or two of the commands may sound a little strange. This is because I have made selection of the desired command easy—you just press the initial letter of the command—but to do so I have kept the commands in a strict alphabetical order and therefore had to find some appropriate way of expressing each command using the available initial letter.

After selecting the action from the main menu the player gets one or more submenus which allow various choices to be made within that broad category of action. For example, in the hunting routine the player can choose which province to hunt in, the weapon to be used, and the tactic to be tried. Success in the hunt depends on all these choices and also on the animals found and the tactics chosen by the animals. This routine actually uses quite a complex algorithm to work out whether the hunt is successful or not, and is a good example of how complexity within the program can make an

an$(7)	animal names
alf	alliance flag
cas$	castle graphic
d%(3,9)	lords records
dep	deposed flag
dft	defeat flag
g	GET variable
ho$	"How many "
h$	GET variable
i	loop control
item	loop control
j	loop control
l%(15,7)	records for each land
lord$(3)	lords´ names
lord	current lord
land	current land
lett$	"ABCDEFGHIJKLMNO"
prov$(15)	land names
pr	prince flag
season$(4)	season names
sh	shift troop flag
sqr	controls printing of squares
sea	season number
tr	treaty flag
we$(7)	weather types
w%	wear on Mernar´s castle
we%	weather
y$	message

```
System variables used for passing parameters
A%, B%, C%, D%, E%, K%
```

Figure 4.3 Variables in Mernar Keep.

apparently simple set of choices available to the player into a testing set of decisions.

Initially the number of animals found is calculated, the probability being determined by the amount of forest in the province. This in turn depends on how much has been burned to make land suitable for crops, and that will depend on how many peasants and men-at-arms the lord has to support. There is always a chance that no animals at all will be found, in which case the lord must still pay for the hunt but has no return on his investment. Before setting out on the hunt the player must choose a weapon. Although it might seem like an arbitrary choice each of the three weapons is best used in a particular way. If the bow is chosen then the appropriate tactic is to stand and wait. If the boar spear is chosen it is best to charge the animal.

Each time an animal is found it will choose its own tactic and the player must choose an appropriate one to counter it. Different animals choose different tactics and, after playing the game for a while, the player will begin to learn which tactics a particular animal is likely to choose. As the tactics chosen by the player are closely related to the tactics chosen by the animal (e.g., if the animal decides to charge it is not very appropriate to attempt to sneak up on it!) as well as on the weapon, and as the efficacy of different weapons is related to the type of animals, choosing a tactic each time you meet an animal can be quite a complicated decision.

Suppose, for example, you have chosen the short spear. You wander into the forest and encounter a boar. From past experience you think boars generally charge but sometimes try to hide. The best tactic with a short spear is to try to sneak up on the creature, but sneaking is not a good way of counteracting a charge. The best way to nullify a charge is to charge in return, but the short spear is only mediocre for charging and what if the boar decides to flee after all? Thus, do you gamble on the more likely tactic of the boar or do you choose the more cautious tactic and hope that circumstances favour you?

If you add to this the fact that the prestige won by a successful hunt depends on the number of animals killed, their type, and the heroic nature of the tactics used, you will see that this section of the program alone requires some intelligent thinking.

Most of the program's sections are self-explanatory. All of the decisions Lord Mernar can make may affect and/or be affected by other aspects of the game. For example, he may decide that he needs more land to increase prestige and crop-planting area. He would thus want to invade an adjacent territory. If he does so he needs an army, which means sufficient men-at-arms and peasants. If he has not enough men-at-arms he may recruit more, but he has to pay them. If he does not have enough money in the treasury he may have to borrow from the King, which will lower his prestige, or he may want to levy a tax on a province. Doing this may, however, cause a riot which might do more harm than good. Similarly he needs to take a large number of peasants to guarantee victory but this will reduce the number able to work in the fields and consequently the harvest may be poor. He must also consider the weather as this will affect fighting and, if he has no alliance with either of the lords of Hale and Arden, whether they are allied, because this may mean the enemy is too strong or might even take retaliatory action against him.

The lords of Hale and Arden are supposedly carrying out the same kind of decisions but we do not have enough memory to cover that. Instead, each lord will carry out one or more actions each season, the number of actions depending to some extent on the season, which may include hostility to Mernar but may be simple recruitment or planting and might even involve hostility between each other.

At any time from the main menu Mernar can choose to end the season or end the game, to get a map of current ownership of the territories, or to examine his scribe's records of his current status. Remember that the sole object of the game is to increase prestige and that everything else is subordinate to this one aim. Prestige is a function of the number of peasants on the lands, the size of the army, the success and/or failure of hunts, the number of territories owned, the number of victories, and alliance with one of the lords. If your prestige exceeds that of one of the lords by a set amount you will automatically gain his lands, and if you can achieve a prestige double that of either lord you will be made Prince of the Isle, gain permanent rule, and win the game. This is not easy.

4.2 Some additional notes on Mernar Keep

Additional points of interest to bear in mind when designing a game like Mernar Keep (Fig. 4.4), which depends very much on variety and internal complexity, are the following:

1. Because the game manipulates so much information the memory consumed is important. Mode 4 has been used because it only uses 10K of memory for graphics but allows an attractive 40-column screen. Graphics are necessary in order to display the map, but colour is not needed as patterns can be used to indicate the different provinces. Similarly, it would be nice to make the displays more interesting with colour but mode 4 only allows two colours at a time. Consequently I have varied the background for different displays to make it a little more interesting visually.
2. The DATA statements at the end of the program are a particularly wasteful kind of programming. Putting one DATA item on a line wastes several bytes of precious memory. However, I have done this to make the organization of the DATA clearer. For each province the first line of DATA holds the number of 'squares' that comprise that province, the second line is the start position for drawing each square, the third line is a start position for drawing the province outline, the fourth is the number of lines in the outline, the fifth holds the coordinates for the outlines, and the sixth holds the province name and the position it is to be printed at. These are READ into the drawing routines each time PROCmap is called. Astute programmers will realize (1) that some of this information is redundant and the drawing routine could be rewritten to dispense with some of this DATA and (2) drawing the map would be faster if the DATA did not have to be

read in each time but was permanently held as part of the routine itself. The drawback with carrying out the first of these modifications is that the program becomes more complex and less easy to understand, and with the second more memory is needed because the commands MOVE, PLOT, and DRAW would be used many times over and the same basic procedures would be carried out for each province drawn. However, you might like to experiment with these variations.

3. The easiest area to expand in order to make the game even more interesting would be the random event routines. It would be a relatively simple matter to add more possible random events to the potential actions each season by simply writing more PROCedures and calling them by increasing the range of the random variable at line 4230. I suggest you do this as a way of exploring what can be done in a design of this kind. However, bear two things in mind. Firstly, if your random routine is to be any length at all you will probably need to gain extra memory for it by the methods described above or by reducing some other aspect of the program. For example, the map could be dispensed with and the whole program use mode 6 (gaining at least 2K) if PROCmap were replaced by a list of the territories owned by each lord at any given time.

   Secondly, inclusion of too many random events may have the effect of unbalancing the game. As it stands it contains a fair degree of randomness but not so much that it can cause the player to lose the game (except once in a 'blue moon'). However, if a large number of random events are added their effect may be so great that the decisions available to the player are insignificant in comparison. The player may be interested to see what happens as a result of a series of purely random activities but is more likely to become bored with a game over which there is no control and which does not seem to be affected by any kind of personal skill.

```
 9  REM ***    MERNAR KEEP    ***
10 MODE 4
15 VDU 19,1,7,0,0,0
16 VDU 19,0,4,0,0,0
17 PRINT TAB(13,12);"MERNAR KEEP"
20 PROCinit
25 PROCsc
28
29  REM Main Loop - Supervisor
30 REPEAT
40   PROCweath
50   REPEAT
60     PROCmenu
```

Figure 4.4. Mernar Keep (*continues*)

```
  70      PROCselect
  80      UNTIL g = 76 OR g = 81
  90    IF g<>81 THEN PROCe(sea)
 100    sea = (sea MOD 4)+1
 110    UNTIL dep OR (pr>0) OR g = 81
 118
 119
 120  PROCfarewell
 990 END
 998
 999
1000 DEF PROCcash
1010 PRINT "How much?"
1020 INPUT g%
1030 r = RND(g%)
1040 sd = FNsoc(3,alf)
1050 r = r+(sd*10)
1060 IF r<(g%-(g%/3)) THEN PRINT ´"That´s an
insult" : PROCpause : ENDPROC
1070 PRINT "That´s acceptable"
1080 cor = sd+1
1090 PROCcoff(g%,1)
1095 ENDPROC
1098
1099
1100 DEF PROCmar
1110 IF d%(3,9) = 0 OR d%(alf,9) = 0 THEN PRINT
´"There are no marriageable children" :
PROCpause : ENDPROC
1118
1120 r = RND(5)
1130 PRINT ´"Will you include a dowry?"
1140 REPEAT
1150   yn$ = GET$
1160   UNTIL INSTR("YNyn",yn$)<>0
1170 IF yn$ = "y" OR yn$ = "Y" THEN INPUT "How
much ?"K% : PROCcoff(K%,1)
1180 IF r = 5 THEN PRINT "That has angered the
lord" : ENDPROC
1190 IF r = 4 THEN PRINT "The lord is not
interested" : cor = 1 : ENDPROC
1200 PRINT "A successful agreement"
1210 PROCord(3,9,-1) : PROCord(alf,9,-1)
1220 tr = alf
1230 cor = r
1240 ENDPROC
1248
1249
1300 DEF PROCg
1310 PROCwh
1320 l%(land,1) = alf
1330 cor = RND(3)+3
1340 ENDPROC
1400 DEF PROCanger
```

```
 1410 r = RND(2)
 1420 PRINT "In revenge the lord ";
 1430 IF r = 1 THEN PRINT "invades one of your
lands" : PROCinvade : ENDPROC
 1440 IF r = 2 THEN PRINT "raids one of your
provinces" : PROCraid : ENDPROC
 1450 ENDPROC
 1458
 1459
 1500 DEF PROCraid
 1505 REPEAT
 1510   land = RND(15)
 1520   UNTIL l%(land,1) = 3
 1525 troop% = RND(d%(lord,4))
 1526 peasant% = RND(d%(lord,3))
 1530 PROCfight(lord,3)
 1535 IF dft = 1 THEN ENDPROC
 1540 l%(land,5) = 0 : l%(land,4) = 0 :
l%(land,3) = RND(l%(land,3))
 1545 PRINT prov$(land);" has been raided, but
the raiders flee"
 1546 PROCpause
 1550 ENDPROC
 1558
 1559
 1600 DEF PROCinvade
 1610 REPEAT
 1615   land = RND(15)
 1620   UNTIL l%(land,1) = 3
 1625 troop% = RND(d%(lord,4))
 1630 peasant% = RND(d%(lord,3))
 1640 FOR j = 1 TO 15
 1645   IF l%(j,1) = lord THEN
PROCand(j,3,-(peasant%/d%(lord,7))) :
PROCand(j,4,-(troop%/d%(lord,7)))
 1650   NEXT
 1655 keep = 13
 1660 sd = FNsoc(lord,3)
 1665 IF sd<1 OR land = keep OR RND(sd) = 1 THEN
PROCfight(lord,3)
 1670 IF dft = 1 THEN PRINT "You beat off an
attack in ";prov$(land) : ENDPROC
 1671 PRINT prov$(land);" is seized." :
l%(land,1) = lord
 1675 d%(lord,7) = d%(lord,7)+1
 1680 d%(3,7) = d%(3,7)-1
 1685 l%(land,1) = lord
 1686 PROCpause
 1690 ENDPROC
 1698
 1699
 1700 DEF PROCshift
 1710 CLS
 1715 sh = 2
```

Figure 4.4. (*continues*)

```
 1720 PROCwh : t = land
 1725 sh = 1
 1730 PROCwh
 1735 sh = 0
 1740 PRINT ´"Move how many of your
";l%(land,3);" peasants?"
 1745 REPEAT
 1750   INPUT p
 1755   UNTIL p<=l%(land,3)
 1760 PRINT ´"Move how many of your
";l%(land,4);" men at arms?"
 1765 REPEAT
 1770   INPUT m
 1775   UNTIL m<=l%(land,4)
 1780 PROCtrans
 1785 ENDPROC
 1788
 1789
 1800 DEF PROCpay
 1801 CLS
 1802  PRINT ´lord$(lord);" is taxing
";prov$(land)
 1803 PROCpause
 1805 py = RND(100)
 1810 d%(lord,6) = d%(lord,6)+py
 1820 brig% = brig%+ INT(py/10)
 1890 ENDPROC
 1900 DEF PROCseed
 1901 CLS : PRINT ´lord$(lord);" is planting in
";prov$(land)
 1902 PROCpause
 1904 IF d%(lord,5)<1 THEN ENDPROC
 1905 z = RND(d%(lord,5))
 1910 l%(land,5) = l%(land,5)+z
 1920 d%(lord,5) = d%(lord,5)-z
 1930 ENDPROC
 1948
 1949
 1950 DEF PROCjoin
 1951 CLS
 1952  PRINT ´lord$(lord);" is recruiting in
";prov$(land)
 1953 PROCpause
 1955 z = RND(l%(land,3))
 1960 l%(land,4) = l%(land,4)+z
 1965 l%(land,3) = l%(land,3)-z
 1970 d%(lord,6) = d%(lord,6)-(z*10)
 1975 IF d%(lord,6)<1 THEN d%(lord,6) = 0 :
PROCord(lord,8,- RND(6))
 1980 ENDPROC
 1998
 1999
 2000 DEF PROCmap
 2005 VDU 19,0,1,0,0,0
```

```
2006 VDU 19,1,7,0,0,0
2010 CLS
2020 FOR dp = 1 TO 15
2040   PROCprovinceprint
2050   NEXT
2060 PROCcas
2080 h = GET
2090 ENDPROC
2098
2099
2100 DEF PROCprovinceprint
2110 RESTORE (9000+(dp*10))
2120 READ loopi
2130 FOR i = 1 TO loopi
2140   READ x,y
2150   MOVE x,y
2160   PROCsq
2170   NEXT i
2175 READ name$
2176 READ name$
2177 VDU 5
2178 MOVE x,y : GCOL 0,1
2179 PRINT STRING$(( LEN(name$)), CHR$(230))
2180 MOVE x,y : GCOL 0,0
2181 PRINT name$
2182 VDU 4
2183 GCOL 0,1
2190 ENDPROC
2198
2199
2200 DEF PROCsq
2205 IF l%(dp,1)>1 THEN 2240
2210 PLOT 1,128,-128
2220 PLOT 0,-128,0
2230 PLOT 1,128,128
2231 ENDPROC
2240 IF l%(dp,1) = 2 THEN 2265
2242 FOR x = 0 TO 3
2245   PLOT 0,32,0
2250   PLOT 25,0,-128
2255   PLOT 0,0,128
2260   NEXT x
2262 ENDPROC
2265 FOR x = 0 TO 3
2270   PLOT 0,0,-32
2275   PLOT 25,128,0
2280   PLOT 0,-128,0
2285   NEXT
2290 ENDPROC
2298
2299
2300 DEF PROCboundary
2305 GCOL 0,1
2310 READ x,y
```

Figure 4.4. (*continues*)

```
 2320 MOVE x,y
 2330 PROCdraw
 2340 ENDPROC
 2348
 2349
 2400 DEF PROCdraw
 2410 READ loopj
 2420 FOR j = 1 TO loopj
 2430   READ x,y
 2440   DRAW x,y
 2450   NEXT
 2460 ENDPROC
 2468
 2469
 2500 DEF PROCcas
 2505 VDU 5
 2510 MOVE 128,960
 2520 PRINT cas$
 2530 MOVE 1140,768
 2540 PRINT cas$
 2550 MOVE 600,192
 2560 PRINT cas$
 2590 VDU 4
 2595 ENDPROC
 2598
 2599
 3000 DEF PROCmenu
 3001 VDU 19,1,7,0,0,0
 3002 VDU 19,0,4,0,0,0
 3005 CLS
 3010 PRINT "Press initial letter of desired
action"
 3020 PRINT TAB(10,3);"Annexe province"
 3030 PRINT TAB(10,5);"Burn forest"
 3040 PRINT TAB(10,7);"Crop planting"
 3050 PRINT TAB(10,9);"Display map"
 3060 PRINT TAB(10,11);"Examine scribe´s
records"
 3070 PRINT TAB(10,13);"Find information"
 3080 PRINT TAB(10,15);"Gather taxes"
 3090 PRINT TAB(10,17);"Hunt"
 3100 PRINT TAB(10,19);"Increase troops"
 3110 PRINT TAB(10,21);"Join alliance"
 3120 PRINT TAB(10,23);"Keep repair"
 3130 PRINT TAB(10,25);"Leave season"
 3140 PRINT TAB(10,27);"Move troops/peasants"
 3150 PRINT TAB(10,29);"Quit"
 3490 ENDPROC
 3498
 3499
 3500 DEF PROCselect
 3510 g = GET
 3511 IF g>81 THEN g = g-32
 3512 IF g = 65 PROCannex
 3514 IF g = 66 PROCforest
 3516 IF g = 67 PROCplant
 3520 IF g = 68 PROCmap
 3530 IF g = 69 PROCsc
 3540 IF g = 70 PROCinfo
 3550 IF g = 71 PROCtax
 3560 IF g = 72 PROChunt
 3570 IF g = 73 PROCrecruit
 3580 IF g = 74 PROCally
 3590 IF g = 75 PROCmaintain
 3600 IF g = 77 PROCshift
 3690 IF g = 76 OR g = 81 PRINT TAB(8,29);"Are
you sure? (Y/N)" : REPEAT : h$ = GET$ : UNTIL
INSTR("YyNn",h$)>0 : IF INSTR("YyNn",h$)>2 THEN
g = 99
 3695  ENDPROC
 3698
 3699
 3800 DEF PROCwh
 3801 CLS
 3805 owner = TRUE
 3806 IF sh = 1 THEN PRINT TAB(8,8);"From "
 3807 IF sh = 2 THEN PRINT TAB(10,8);"To "
 3810 PRINT TAB(13,8);"Which province?"
 3820 PRINT TAB(5,10);"(Press initial letter of
name)"
 3821  PRINT ´"     Province",,"Lord"
 3822 PRINT : FOR k = 1 TO 15 : PRINT "
";prov$(k),,lord$(l%(k,1)) : NEXT
 3824 REPEAT
 3825   land = 0
 3826   REPEAT
 3830     g = GET
 3831     IF g>79 THEN g = g-32
 3832     UNTIL g>64 AND g<80
 3835   g$ = CHR$(g)
 3840   land = INSTR(lett$,g$)
 3880   UNTIL land<>0
 3890 IF l%(land,1)<>3 THEN owner = FALSE
 3895 CLS
 3896  g = 0: sh = 0
 3897 ENDPROC
 3898
 3899
 4000 DEF PROCweath
 4005 CLS
 4006 PRINT TAB(5,5);"It is now ";season$(sea)
 4010 we% = RND(3)+sea-1
 4020 PRINT TAB(5,7);"The weather is ";we$(we%)
 4090 PROCpause
 4095 ENDPROC
 4098
 4099
 4100 DEF PROCe(a)
```

Figure 4.4. (*continues*)

```
4105 CLS
4110 w% = (we%*d%(3,4))+(d%(3,3)/100)+w%
4120 kflag = 0
4122 FOR i = 1 TO 15
4123   IF l%(i,7)>0 THEN l%(i,7) = l%(i,7)+1
4124   IF l%(i,7)>3 THEN l%(i,7) = 0
4125   NEXT
4130 FOR lord = 1 TO 2
4140   PROClord(lord)
4150   NEXT
4160 PROCharvest
4170 FOR i = 1 TO 3
4180   PROCprestige(i)
4185   d%(i,8) = 0
4190   NEXT
4192 PROCchek
4195 PROCpause
4196  ENDPROC
4198
4199
4200 DEF PROClord(lord)
4210 r = RND(sea)
4220 FOR k = 1 TO r
4225   REPEAT
4230     s = RND(8)
4240     land = RND(15)
4250     UNTIL (l%(land,1) = lord AND s>2) OR
(l%(land,1)<>lord AND s<3)
4255   NEXT k
4260 IF s = 1 THEN PROCinvade
4270 IF s = 2 THEN PROCraid
4280 IF s = 3 THEN PROCseed
4290 IF s = 4 THEN PROCmove
4300 IF s = 5 THEN PROCpay
4310 IF s = 6 THEN PROCseed
4320 IF s = 7 THEN PROCseed
4330 IF s = 8 THEN PROCjoin
4390 ENDPROC
4398
4399
4500 DEF PROCmove
4510 IF l%(lord,7)<2 THEN ENDPROC
4515 REPEAT
4520   t = RND(15)
4530   UNTIL l%(t,1) = lord
4535 PRINT ´"Troop movements are reported in
";prov$(t)
4536 PROCpause
4540 p = RND(l%(land,3))
4550 m = RND(l%(land,4))
4552 PROCtrans
4555 ENDPROC
4558
4559
```

```
 4560  DEF PROCtrans
 4565  PROCand(land,3,-p)
 4570 PROCand(t,3,p)
 4580 PROCand(land,4,-m)
 4590 PROCand(t,4,m)
 4595 ENDPROC
 4598
 4599
 4600 DEF PROCharvest
 4605 CLS
 4610 FOR i = 1 TO 15
 4620   IF l%(i,1) = 3 THEN PROCcrop
 4630   NEXT
 4640 PRINT "Total crop is ";crop%
 4650 PROCord(3,5,crop%)
 4660 PRINT "You must feed ";d%(3,3);" peasants"
 4670 PRINT SPC (10);"and ";d%(3,4);" men at
arms"
 4680 PROCord(3,5,(d%(3,3)+(d%(3,4)*2)))
 4690 IF d%(3,5)<0 THEN death$ = "starvation" :
PROCstarve : d%(3,5) = 0
 4700 PRINT "This leaves ";d%(3,5);" sacks of
grain"
 4705 PROCpause
 4710 ENDPROC
 4718
 4719
 4720 DEF PROCstarve
 4730 loss = ABS(d%(lord,5))
 4740 death = INT(loss/20)
 4750 PRINT death;" peasants die of ";death$
 4760 FOR k = 1 TO 15
 4765   IF l%(k,1) = lord THEN l%(k,3) =
l%(k,3)-( INT(death/(d%(lord,7))))
 4770   NEXT k
 4775 PROCord(lord,3,-death)
 4780 ENDPROC
 4788
 4789
 4800 DEF PROCcrop
 4810 y% = (l%(i,5)/( RND(we%))*(l%(i,3)/50))
 4820 crop% = crop%+y%
 4830 ENDPROC
 4838
 4839
 4900 DEF PROCand(A%,B%,C%)
 4910 l%(A%,B%) = l%(A%,B%)+C%
 4920 ENDPROC
 4928
 4929
 4930 DEF PROCord(A%,B%,C%)
 4940 d%(A%,B%) = d%(A%,B%)+C%
 4950 ENDPROC
 4958
```

Figure 4.4. *(continues)*

```
 4959
 5000 DEF PROCcoff(nu,K%)
 5001 CLS
 5005 PRINT "That will cost you ";nu*K%;"
crowns"
 5010 IF nu*K%>d%(3,6) THEN PRINT "You´ll have
to borrow from the king.    He´s not pleased" :
PROCord(3,8,( RND(4))) : d%(3,6) = 0
 5020 d%(3,6) = d%(3,6)-(nu*K%)
 5022 IF d%(3,6)<0 THEN d%(3,6) = 0
 5025 PROCpause
 5030 ENDPROC
 5098
 5099
 5100 DEF PROCfight(at,df)
 5110 r = RND(3)
 5112 dmor = 0
 5115 IF r = 1 AND at<>3 THEN amor = d%(at,1)
 5120 IF r = 1 AND df<>3 THEN dmor = d%(df,1)
 5125 PROCshow
 5126 PRINT ´lord$(at);" attacks ";prov$(land)
 5130 PRINT "Tactical report:"´
 5140 IF r = 1 THEN PRINT "The enemy lord will
personally lead his troops"
 5145 PRINT "The weather is ";we$(we%)
 5150 g$ = ""
 5152 amor = 0
 5155 PRINT "Will you lead your troops
personally (Y/N)?"
 5160 g$ = GET$
 5165 IF (g$ = "Y" OR g$ = "y") AND at = 3 THEN
amor = d%(3,1)
 5170 IF (g$ = "Y" OR g$ = "y") AND df = 3 THEN
dmor = d%(3,1)
 5175 PRINT "Will you :"
 5180 PRINT "   (1) Fight defensively"
 5185 PRINT "   (2) Try to outflank the enemy "
 5190 PRINT "   (3) Launch a frontal attack"
 5195 g$ = GET$ : IF VAL(g$)<1 OR VAL(g$)>3 THEN
5195
 5196 IF at = 3 THEN atac = VAL(g$) ELSE dtac =
VAL(g$)
 5197 IF df<>3 THEN dtac = RND(3) ELSE atac =
RND(3)
 5200 att = (troop%+((
RND(peasant%/10))*10))*((amor+atac)/5)-(we%*6)+(
fd%/atac)
 5205 def = (l%(land,4)+(
RND(l%(land,3)/8)*8))*((dmor+dtac)/6)+(fd%/dtac)
 5220 IF att<=def THEN dft = 1 : PRINT "The
attacker loses" : PROCpause : ENDPROC
 5225 IF (at = 3 AND amor<>0 AND def>att) OR (df
= 3 AND dmor<>0 AND att>def) THEN p = RND(5) :
```

```
PRINT "You lose ";p;" prestige from the battle"
: d%(3,8) = d%(3,8)-p
 5226 IF (at = 3 AND amor<>0 AND att>def) OR (df
= 3 AND dmor<>0 AND def>att) THEN p = RND(5) :
PRINT "You win ";p;" prestige from the battle" :
PROCord(3,8,p)
 5227 IF att>defTHENPRINT´"The attacker wins"
ELSE PRINT "The defender wins"
 5230 l%(land,4) = l%(land,4)+troop%-(
RND(l%(land,4)+troop%)* RND(2))
 5235 IF l%(land,4)<0 THEN l%(land,4) = 0
 5240 l%(land,3) = l%(land,3)+peasant%-(
RND(l%(land,3)+peasant%)* RND(2))
 5245 IF l%(land,3)<0 THEN l%(land,3) = 0
 5250 PROCpause
 5290 ENDPROC
 5298
 5299
 5300 DEF PROCpause
 5320 PRINT : PRINT "    (Press any key to
continue)"
 5325 G = GET
 5330 ENDPROC
 5398
 5399
 5400 DEF PROCriot
 5405 CLS
 5410 PRINT ´´"The peasants of ";prov$(land);"
rebel"
 5420 m = RND(l%(land,4))
 5425 PRINT ´"They linch ";m;" men at arms"
 5430 l%(land,4) = l%(land,4)-m
 5435 p = RND(l%(land,3))
 5440 g = INT( RND(l%(land,5)))
 5445 IF g>0 THEN PRINT ´"And steal ";g;" sacks
of grain"
 5450 PRINT ´"Losing ";p;" lives in the
fighting"
 5455 l%(land,3) = l%(land,3)-p
 5460 l%(land,5) = l%(land,5)-g
 5485 PROCpause
 5490 ENDPROC
 5498
 5499
 5500 DEF PROCshow
 5505 CLS
 5510 l = LEN(prov$(land)) DIV 2
 5520 PRINT TAB(20-l,2);prov$(land)
 5530 PRINT ´"Cultivated land ";l%(land,2)+
FNspy(l%(land,2));" acres"
 5540 PRINT ´"Peasants ";l%(land,3)+
FNspy(l%(land,3))
 5550 PRINT ´"Men at arms ";l%(land,4)+
FNspy(l%(land,4))
```

Figure 4.4. *(continues)*

```
 5560 PRINT ´"Current crop ";l%(land,5)+
FNspy(l%(land,5))
 5565 PRINT
 5570 IF l%(land,7)>0 THEN PRINT "Already taxed
this year" : PRINT
 5580 PROCtree
 5590 ENDPROC
 5598
 5599
 5600 DEF PROCtree
 5610 RESTORE (9000+(land*10))
 5620 READ sqr
 5630 fd% = (sqr*100)-l%(land,2)
 5640 PRINT ´;"There are ";fd%;" acres of forest
left."
 5644 *FX 15,1
 5645 g = INKEY(200)
 5650 ENDPROC
 5658
 5659
 5700 DEF PROCprestige(lord)
 5710 d%(lord,1) =
INT((d%(lord,3)/1000)+d%(lord,4)/250)+(d%(lord,1
)/4)+(d%(lord,7))+tr+(d%(lord,6)/100)+d%(lord,8)
 5715 IF lord = 3 THEN PROCord(lord,1,-(
INT(w%/100)))
 5780 PRINT "  ";lord$(lord);"´s prestige is
";d%(lord,1)
 5790 ENDPROC
 5798
 5799
 5800 DEF PROCchek
 5810 FOR k = 1 TO 3
 5815   IF l%(13,1)<>3 THEN dep = TRUE : x$ =
"you have lost your castle"
 5820   IF d%(3,1)<1 THEN dep = TRUE : x$ = "
your prestige has fallen so low."
 5830   IF d%(3,5)<1 THEN dep = TRUE : x$ = "
you own no lands."
 5840   IF d%(3,7)<1 THEN dep = TRUE : x$ = "
you cannot feed your people."
 5845   hi = 0 : lo = 1
 5850   FOR k = 1 TO 3
 5860     IF d%(k,1)>hi THEN hi = d%(k,1)
 5870      IF d%(k,1)<lo THEN lo = d%(k,1)
 5885     IF (hi-lo)>19 THEN pr = k
 5890     NEXT
 5900   ENDPROC
 5998
 5999
 6000   DEF PROCannex
 6010   PROCwh
 6020   IF owner = TRUE THEN PRINT ´´"    It´s
yours already" : PROCpause : ENDPROC
```

```
 6025   REPEAT
 6030     IF d%(3,4)>0 THEN PRINT
TAB(5,5);ho$;"men will you use?"
 6040     INPUT troop%
 6045     UNTIL troop%<=d%(3,4)
 6046   REPEAT
 6050     IF d%(3,3)>0 THEN PRINT
TAB(5,7);ho$;"peasants will you use?"
 6060     INPUT peasant%
 6065     UNTIL peasant%<=d%(3,3)
 6066   FOR j = 1 TO 15
 6067     IF l%(j,1) = 3 THEN
PROCand(j,3,(-peasant%/d%(3,7))) :
PROCand(j,4,(-troop%/d%(3,7)))
 6070     NEXT
 6075   enemy = l%(land,1)
 6080   IF enemy = 1 THEN keep = 1 ELSE keep = 8
 6090   sd = FNsoc(3,enemy)
 6100   IF sd<1 OR land = keep OR RND(sd) = 1
THEN PROCfight(3,l%(land,1))
 6110   IF dft = 1 THEN PRINT TAB(10,0);"You are
soundly defeated" : PROCpause : ENDPROC
 6115    PRINT "You seize ";prov$(land)
 6120   l%(land,1) = 3
 6130   PROCord(3,7,1)
 6135   d%(l%(land,1),7) = d%(l%(land,1),7)-1
 6140   PROCpause
 6190   ENDPROC
 6198
 6199
 6200   DEF PROCforest
 6210   PROCwh
 6215   IF owner = FALSE THEN PRINT y$ :
PROCpause : ENDPROC
 6216   PROCshow
 6230   REPEAT
 6260     PRINT : PRINT ho$;"acres will you
reclaim?" : INPUT acre%
 6270     UNTIL acre%<=fd%
 6280   PROCand(land,2,acre%)
 6285   PROCord(3,2,acre%)
 6390   ENDPROC
 6398
 6399
 6400   DEF PROCplant
 6410   PROCwh
 6412   IF owner = FALSE THEN PRINT y$ :
PROCpause : ENDPROC
 6413   PROCshow
 6414   IF d%(3,5)<1 THEN PRINT ´"You´ve nothing
to plant" : ENDPROC
 6415   REPEAT
 6416     REPEAT
 6420       PRINT TAB(4,16);ho$;"of your
```

Figure 4.4.
(continues)

```
 ";d%(3,5);" sacks of grain will you plant?"
 6430        INPUT n
 6435        UNTIL n<=d%(3,5)
 6440      IF n DIV 100>l%(land,2) THEN PRINT
"Not enough land to plant this"
 6450      UNTIL n DIV 100<=l%(land,2)
 6455    PROCord(3,5,-n)
 6480    l%(land,5) = n
 6590    ENDPROC
 6600    DEF PROCsc
 6602    CLS
 6603    FOR item = 2 TO 4
 6604      d%(3,item) = 0
 6605      FOR land = 1 TO 15
 6608        PROCcount(3,item,land)
 6609        NEXT : NEXT
 6610    PRINT : PROCprestige(3)
 6620    PRINT ´"  Coffers contain ";d%(3,6);"
crowns"
 6630    PRINT ´"  Lands owned ";d%(3,7)
 6640    PRINT ´"  Men at arms ";d%(3,4)
 6650    PRINT ´"  Peasants ";d%(3,3)
 6660    PRINT ´"  Grain ";d%(3,5);" sacks"
 6670    PRINT ´"  Total farming land ";d%(3,2);"
acres"
 6690    PROCpause
 6790    ENDPROC
 6798
 6799
 6800    DEF PROCally
 6805    CLS
 6810    IF alf<>0 THEN PRINT TAB(0,10);"No-one
will admit your ambassador" : PROCpause :
ENDPROC
 6815    REPEAT
 6820      PRINT TAB(0,10);"Do you want to ally
with Ardan(1)"´"                          or Hale
(2) ?"
 6830      alf = VAL( GET$)
 6835      UNTIL alf>0 AND alf<3
 6855    IF alf = tr THEN PRINT "You already have
an alliance with this lord" : PROCpause :
ENDPROC
 6860    IF tr>0 THEN PRINT "You are allied with
the other lord. He´s angry about your treachery
and plans to get his revenge" : tr = 0 :
PROCpause : PROCinvade : ENDPROC
 6865    PRINT "Do you want alliance by marriage
 (1)"
 6870    PRINT SPC (20);" gift of land  (2)"
 6875    PRINT SPC (20);" gift of money (3)"
 6876    REPEAT
 6880     t = VAL( GET$)
 6882     UNTIL t>0 AND t<4
 6885   cor = 0
 6890   IF t = 1 THEN PROCmar
 6895   IF t = 2 THEN PROCg
 6900   IF t = 3 THEN PROCcash
 6905   IF cor = 0 THEN PROCanger : ENDPROC
 6906    IF cor = 1 THEN ENDPROC
 6910   PRINT "You have a cordial alliance"
 6915   tr = alf
 6920   PROCpause
 6925   ENDPROC
 6998
 6999
 7000   DEF PROCtax
 7010   PROCwh
 7012   IF owner = FALSE THEN PRINT y$ :
PROCpause : ENDPROC
 7015   CLS
 7016   PROCshow
 7020   IF l%(land,7)>0 THEN PROCriot : ENDPROC
 7030   PRINT ´"What is the levy for this
province?"
 7040   INPUT crowns
 7050   growa = l%(land,2)-((l%(land,5)) DIV
100)+1
 7060   subsist = (growa/(l%(land,3)))*5
 7070   ptax = (crowns/l%(land,3))*10
 7080   IF ptax>subsist THEN PROCriot : ENDPROC
 7090   tpay = INT(crowns*(subsist-ptax))
 7095   IF tpay<1 THEN tpay = 0
 7096   IF tpay>crowns THEN tpay = crowns
 7100   PRINT ´"  You collect ";tpay;" crowns"
 7110   PROCord(3,6,tpay)
 7120   brig% = brig%+tpay DIV 5
 7130   PROCand(land,3,(tpay DIV 5))
 7135   l%(land,7) = 1
 7140   PROCpause
 7151   CLS : PRINT ´"Here comes something else"
 7190   ENDPROC
 7198
 7199
 7200   DEF PROChunt
 7205   hunt = FALSE
 7210   PROCwh
 7215   IF owner = FALSE THEN PRINT y$ :
PROCpause : ENDPROC
 7240   PROCtree
 7244   count = 0
 7245   antot = RND( INT(fd%/120))
 7246   IF antot<1 THEN antot = 0 : PRINT "A
wasted expedition, you´ve destroyed all the
wildlife here" : PROCpause : ENDPROC
 7250   REPEAT
```

Figure 4.4. (*continues*)

```
 7251      CLS
 7252      PRINT ´"What weapon will you take?"
 7253      PRINT ´"Bow (1)"
 7254      PRINT ´"Short spear (2)"
 7255      INPUT ´"Boar spear (3) ",wpn
 7256      count = count+1 : an% = ( RND(6)+
RND(8)) DIV 2
 7257      IF an%>6 THEN PRINT ´" A wasted
journey" : hunt = TRUE
 7260      CLS : PRINT ´"You find a ";an$(an%)
 7270      PRINT ´"Do you want to ambush it (1)"
 7275      PRINT ´"Sneak up on it (2)"
 7280      INPUT ´"or charge it (3)",hutac
 7295      IF an%>= RND(8) THEN tactic = 3 ELSE
IF (7-an%)>= RND(8) THEN tactic = 2 ELSE tactic
= 1
 7305      diff = ABS(hutac+tactic-4)
 7315      IF (wpn = 3 AND hutac = 2) THEN effect
= 2 ELSE effect = ABS(wpn-hutac)
 7325      kill = 2
 7335      IF ((an%+2) DIV 3 = 1 AND wpn = 1) OR
((an%+2) DIV 3 = 3 AND wpn = 3) THEN kill = 1
 7345      IF 4-((an%+1) DIV 2) = wpn THEN kill =
3
 7355      kill = kill+diff+effect
 7365      pres = -1
 7370      IF kill> RND(7) THEN PRINT ´"The
creature is too fast for you" ELSE PRINT ´"You
slay the ";an$(an%) : pres = 1 :
PROCord(3,5,an%*10)
 7374      PROCord(3,8,pres)
 7375      PRINT ´"Your prestige bonus this
season is now ";d%(3,8)
 7376      PROCpause
 7380      UNTIL hunt = TRUE OR count = antot
 7385    K% = antot*wpn
 7386    PROCcoff(1,K%)
 7390    ENDPROC
 7398
 7399
 7400    DEF PROCrecruit
 7410    CLS
 7420    PRINT TAB(0,3);ho$;"troops do you wish
to recruit?"
 7430    INPUT n
 7440    dec% = n/d%(3,7)
 7450    IF dec%<1 OR dec%>d%(3,7)*100 THEN PRINT
"Not possible" : PROCpause : ENDPROC
 7460    FOR i = 1 TO 15
 7470      IF l%(i,1) = 3 THEN PROCand(i,3,-dec%)
: PROCand(i,4,dec%)
 7480      NEXT i
 7490    PROCcoff(n,10)
 7590    ENDPROC
 7598
 7599
 7600   DEF PROCinfo
 7605   CLS
 7610   PRINT TAB(0,10);"Information on your own
lands is correct"´," but on other lands may be
distorted by"´´; SPC (15);" your spies"´
 7615   PROCpause
 7620   PROCwh
 7630   PROCshow
 7680   PROCpause
 7790   ENDPROC
 7798
 7799
 7800   DEF PROCmaintain
 7820   charge% = RND(3)
 7830   CLS
 7840   PRINT TAB(4,10);"For full repairs it
will cost ";w%*charge%
 7845   PRINT TAB(4,12);
 7850   INPUT "How much will you pay ",n
 7860   PROCcoff(n,1)
 7870   w% = w%-(n/charge%)
 7880   IF w%<0 THEN w% = 0
 7890   ENDPROC
 7898
 7899
 7900   DEF PROCcount(lord,item,land)
 7910   IF l%(land,1) = lord THEN
PROCord(lord,item,l%(land,item))
 7920   kflag = 1
 7930   ENDPROC
 7938
 7939
 8000   DEF PROCinit
 8010   VDU 23,225,&E0A0;&A0E0;&BFB5;&FBFB;
 8020   VDU 23,226,&0705;&0507;&FDAD;&DFDF;
 8030   VDU 23,227,&FFFB;&BEBF;&FCBC;&FCFC;
 8040   VDU 23,228,&FFDF;&7DFD;&3F3D;&3F3F;
 8050   VDU 23,230,&FFFF;&FFFF;&FFFF;&FFFF;
 8100   cas$ = CHR$(225)+ CHR$(226)+ CHR$(8)+
CHR$(8)+ CHR$(10)+ CHR$(227)+ CHR$(228)
 8120   DIM
l%(15,7),d%(3,9),prov$(15),we$(7),an$(7),season$
(4),lord$(3)
 8125   sh = 0
 8126   ho$ = "How many "
 8130   FOR i = 0 TO 14
 8140     l%(i+1,1) = (i DIV 5)+1
 8142     RESTORE (9000+((i+1)*10))
 8143     READ sqr
 8144     l%(i+1,2) = sqr* RND(6)
 8146     l%(i+1,3) = RND(300)+200
 8147     l%(i+1,4) = RND(30)+10
```

Figure 4.4. (*continues*)

```
 8150      FOR j = 1 TO 3
 8152        PROCcount((i DIV 5)+1,j+1,i+1)
 8153        d%(j,9) = RND(4)
 8154        d%(j,1) = RND(8)+1
 8155        d%(j,5) = RND(250)+250
 8156        d%(j,6) = RND(50)+50
 8157        d%(j,7) = 5
 8160        NEXT : NEXT
 8161    RESTORE 9000
 8163    FOR i = 1 TO 4
 8164      READ season$(i)
 8165      NEXT
 8166    FOR i = 1 TO 3 : READ lord$(i) : NEXT
 8170    alf = 0 : dep = FALSE : pr = 0
 8175    kflag = 0 : sea = 1
 8180    lett$ = "ABCDEFGHIJKLMNO"
 8182    FOR i = 1 TO 15
 8186      READ prov$(i)
 8187      NEXT
 8190    w% = 1 : dft = 0 : tr = 0
 8192    FOR i = 1 TO 7
 8194      READ we$(i),an$(i)
 8196      NEXT
 8200    y$ = CHR$(10)+ CHR$(10)+"   You don´t
own that territory"
 8290    ENDPROC
 8298
 8299
 8300     DEF PROCfarewell
 8310    CLS
 8320    IF dep = TRUE THEN PRINT "You are
deposed because ";x$
 8330    IF pr>0 THEN PRINT lord$(pr);" is made
Prince of the realm."
 8340    PRINT ´"      Farewell"
 8350    ENDPROC
 8358
 8359
 8360    DEF FNspy(x)
 8370    LOCAL D%,E%
 8380    D% = RND(2)
 8390    IF D% = 2 THEN D% = -1
 8400    E% = ( RND(x/4)*D%)
 8410    IF l%(land,1) = 3 THEN E% = 0
 8420    = E%
 8428
 8429
 8990    DEF FNsoc(x,y) = d%(x,1)-d%(y,1)
 8998     REM ***** DATA *****
 8999
 9000    DATA
spring,summer,autumn,winter,Ardan,Hale,Mernar
 9001    DATA
Ardan,Belar,Cantour,Dean,Ellyn,Fier,Gast,Hale,Ir
el,Jeren,Kath,Lirellan,Mernar,Noth,Oran
```

```
 9005   DATA
sunny,badger,sunny,deer,dry,fox,damp,stag,very
wet,wolf,stormy,boar,very stormy,nothing
 9009    REM DATA FOR ARDAN
 9010   DATA 6
 9011   DATA
0,1023,128,1023,256,1023,0,896,128,896,256,896
 9012   DATA 0,1023
 9013   DATA 4
 9014   DATA 384,1023,384,768,0,768,0,1023
 9015   DATA Ardan,56,864
 9019    REM DATA FOR BELAR
 9020   DATA 4
 9021   DATA 384,1023,384,896,512,896,640,896
 9022   DATA 384,1023
 9023   DATA 6
 9024   DATA
512,1023,512,896,768,896,768,768,384,768,384,102
3
 9025   DATA Belar,464,864
 9029    REM DATA FOR CANTOUR
 9030   DATA 7
 9031   DATA
0,768,128,768,256,768,0,640,128,640,256,640,256,
512
 9032   DATA 0,768
 9033   DATA 6
 9034   DATA
384,768,384,384,256,384,256,512,0,512,0,768
 9035   DATA Cantour,64,640
 9039    REM DATA FOR DEAN
 9040   DATA 4
 9041   DATA 384,768,512,768,384,640,512,640
 9042   DATA 384,768
 9043   DATA 4
 9044   DATA 640,768,640,512,384,512,384,768
 9045   DATA Dean,464,640
 9049    REM DATA FOR ELLYN
 9050   DATA 6
 9051   DATA
0,512,128,512,0,384,128,384,256,384,0,256
 9052   DATA 0,512
 9053   DATA 8
 9054   DATA
256,512,256,384,384,384,384,256,128,256,128,128,
0,128,0,512
 9055   DATA Ellyn,64,320
 9059    REM DATA FOR FIER
 9060   DATA 6
 9061   DATA
512,1024,640,1024,768,1024,896,1024,768,896,896,
896
 9062   DATA 512,1023
 9063   DATA 6
```

Figure 4.4. (*continues*)

```
 9064   DATA
1024,1023,1024,768,768,768,768,896,512,896,512,1
023
 9065   DATA Fier,800,864
 9069    REM DATA FOR GAST
 9070   DATA 5
 9071   DATA
640,768,768,768,898,768,768,640,898,640
 9072   DATA 640,768
 9073   DATA 6
 9074   DATA
1024,768,1024,512,768,512,768,640,640,640,640,76
8
 9075   DATA Gast,800,640
 9079    REM DATA FOR HALE
 9080   DATA 6
 9081   DATA
1024,1024,1152,1024,1024,896,1152,896,1024,768,1
152,768
 9082   DATA 1024,1023
 9083   DATA 4
 9084   DATA
1279,1023,1279,640,1024,640,1024,1023
 9085   DATA Hale,1100,864
 9089    REM DATA FOR IREL
 9090   DATA 4
 9091   DATA 1024,640,1152,640,1024,512,1152,512
 9092   DATA 1024,640
 9093   DATA 4
 9094   DATA 1279,640,1279,384,1024,384,1024,640
 9095   DATA Irel,1100,512
 9099    REM DATA FOR JEREN
 9100   DATA 5
 9101   DATA
896,384,1024,384,1152,384,1152,256,1152,128
 9102   DATA 896,384
 9103   DATA 6
 9104   DATA
1279,384,1279,0,1152,0,1152,256,896,256,896,384
 9105   DATA Jeren,1000,320
 9109    REM DATA FOR KATH
 9110   DATA 4
 9111   DATA 384,512,512,512,384,384,512,384
 9112   DATA 384,512
 9113   DATA 4
 9114   DATA 640,512,640,256,384,256,384,512
 9115   DATA Kath,464,384
 9119    REM DATA FOR LIRELLAN
 9120   DATA 7
 9121   DATA
128,256,256,256,384,256,0,128,128,128,256,128,38
4,128
 9122   DATA 0,128
 9123   DATA 6
 9124   DATA
128,128,128,256,512,256,512,0,0,0,0,128
 9125   DATA Lirellan,200,128
 9129    REM DATA FOR MERNAR
 9130   DATA 4
 9131   DATA 512,256,640,256,512,128,640,128
 9132   DATA 512,256
 9133   DATA 4
 9134   DATA 768,256,768,0,512,0,512,256
 9135   DATA Mernar,540,128
 9139    REM DATA FOR NOTH
 9140   DATA 6
 9141   DATA
768,256,896,256,1024,256,768,128,896,128,1024,12
8
 9142   DATA 768,256
 9143   DATA 4
 9144   DATA 1152,256,1152,0,768,0,768,256
 9145   DATA Noth,896,128
 9149    REM DATA FOR ORAN
 9150   DATA 6
 9151   DATA
640,640,640,512,640,384,768,512,768,384,896,512
 9152   DATA 640,640
 9153   DATA 8
 9154   DATA
768,640,768,512,1024,512,1024,384,896,384,896,25
6,640,256,640,640
 9155   DATA Oran,704,384
```

Figure 4.4 Mernar Keep

5 INTELLIGENT STRATEGY

5.1 Games that learn

If you were playing Noughts and Crosses and each time your opponent placed his cross in the centre square he won the game, in subsequent games you might try putting your nought in the centre, either to stop him winning or in the hope that you would win. If you were playing Scissors, Paper, Stone and your opponent always used 'stone' you would soon learn to use 'paper' every time, because paper always beats stone. Whenever you play a game you are continually learning. You learn how the game should be played in order to give yourself a better chance of winning (as in the Noughts and Crosses example) and you learn how your opponent plays so you can adapt your own play accordingly (as in the second example). This is intelligent play. There is little point in ignoring your opponent or the nature of the game if you want to win. You must learn how to play, how to play well, and how to play well against different kinds of players.

One criterion for an intelligent game might be that it should be able to learn. We will know that it is learning when its play generally improves or if its strategy is constantly updated to match that of the player. Thus in Scissors, Paper, Stone one would expect a program to win about 50 per cent of the time, but less if the human player is employing an intelligent strategy. For example, if our micro version of the game simply chose randomly each turn from the three available alternatives all the human player needs to do is to play the same choice every time to ensure, in the long term, that both players come out even. Alternatively, as there is always a greater chance that the program will select to play a choice different from the previous choice rather than the same choice, if the player selects that choice he or she should win in the long term.

Let us look at a program to play this simple game and find out how it might learn to play against a particular opponent. The program is very simple and is given in Fig. 5.1. The player chooses one of the three moves Rock, Scissors, or Tissue and the computer independently makes its own choice. Rock beats Scissors, Scissors beats Tissue, and Tissue beats Rock. Every time the player makes a

```
 10 ON ERROR GOTO 540
 20 MODE1
 30 COLOUR 129
 40 CLS
 50 PRINT''''
 60 PRINT "    GUESS ROCK, SCISSORS OR TISSUE"
 70 PRINT"    BY PRESSING THE INITIAL LETTER"
 80 PRINT'';"    TO END GAME PRESS <ESCAPE>"
 90 PROCwait
100 CLS
110 PROCinit
120
130 REM Main Loop
140 REPEAT
150  PRINTTAB(0,0);"MY TOTAL ";
160  COLOUR 2
170  PRINTTAB(10,0);total(1)
180  COLOUR 3
190  PRINTTAB(20,0);"YOUR TOTAL ";
200  COLOUR 2
210  PRINTTAB(32,0);total(2)
220  COLOUR 3
230   REPEAT
240   PRINTTAB(29,10);"        " :
      PRINTTAB(11,10);"        ";
      TAB(12,15);"             ";TAB(12,16);"             "
250   move=1+((moves(s,1)<=moves(s,2))^2)
260   move=move+((moves(s,move)<=moves(s,3))^2)*(3-move)
270     r=s
280     move=(move+3*((1=move)^2))-1
290     PRINTTAB(0,10);"YOUR GUESS ";
300       REPEAT
310       guess$=GET$
320       UNTIL INSTR("RST",guess$)<>0
330      REPEAT
340      X=0
350      PROCupdate
360      UNTIL X<>0
370     PRINTMID$(possible$,X,8)
380     IF guess$="P" THEN s=1 ELSE s=ASC(guess$)-81
390     PRINTTAB(20,10);"MY GUESS ";
400     guess$=CHR$(move+81)
410     X=0
420     PROCupdate
430     PRINTMID$(possible$,X,8)
440     moves(r,s)=moves(r,s)+1
450     IF s=move THEN PRINTTAB(15,16);"A DRAW":PROCwait
460   UNTIL s<>move
470   win=s-move+3*((move>s)^2)
480   total(win)=total(win)+1
490   PRINT TAB(12,16);name$(win);" WIN"
500   PROCwait
510 UNTIL FALSE
520
```

```
530
540 REM End Routine
550 CLS
560 PRINT´´´´´
570 IF total(1)>=total(2) THEN
    PRINT"It looks as if I´m too good for you"
    ELSE PRINT"I suppose you think you´re clever"
580 END
590
600
610 DEFPROCupdate
620 FOR i=1 TO 24 STEP 8
630 IF guess$=MID$(possible$,i,1) THEN X=i
640   NEXT i
650 ENDPROC
660
670
680 DEFPROCinit
690 possible$="ROCK    SCISSORSTISSUE   "
700 DIM total(2), moves(3,3), name$(2)
710 r=INT(RND(3))
720 s=INT(RND(3))
730  moves(r,s)=1
740  moves(s,r)=1
750  name$(1)="I"
760  name$(2)="YOU"
770 ENDPROC
780
790
800  REM Waits until space bar is pressed
810  DEFPROCwait
820  PRINT TAB(5,20);"Press space bar to continue"
830  REPEAT
840    g=GET
850    UNTIL g=32
860 PRINTTAB(5,20);STRING$(30," ")
870 ENDPROC
```

Figure 5.1 'Scissors'.

move the computer remembers it, together with the player's previous move. Thus it 'knows' how many times playing Rock has been followed by playing Tissue, for example. This is the purpose of the 3 × 3 array. For each of the three possible moves that could be played it holds the number of times that they have been played and the number of times they have been followed by each kind of move. Thus when it has to make its own choice it looks at the move the player last made, looks along the table to see which move most frequently has been used by the player to follow that move, and plays the appropriate counter-move.

Suppose the player always follows Rock by playing Scissors. The program will have in its array the information “Rock followed by rock—0”, “Rock followed by Scissors—8”, and “Rock followed by Tissue—0”. So it ‘knows’ that Scissors is the likely choice and will thus choose Rock itself. As soon as the player realizes that this is what is happening another strategy will be chosen, but as soon as the player has used this strategy more than the other one the computer will also switch strategies.

In this way the program is always learning about the player and if the player sticks to a rigid routine the computer will always win. Of course, most players realize this and so try to play as randomly as possible. It is unlikely that a player would lose to this program more than once or twice. However, the result of this simple routine is that the player must constantly be thinking about the game and must play intelligently in order to win.

How could we improve the program so that it might have a better chance? The best way would be to set up larger arrays that remembered not just the previous move but the previous two, three, four, or more moves and looked for patterns in play. However, the program would also have to incorporate what is known as an ‘evaluative function’ which might be quite complex. An evaluative function in a game is a single function which gives weight to all the possible variables and aspects of the situation and combines them in a way which the programmer thinks shows the best course to take, usually by resulting in a single number which controls the choice of the program’s move.

For example, if our program remembered patterns of two moves and also patterns of three we might have an evaluative function which regarded the two-move pattern as more important than the longer one, so gave a weighting of 2 to the former and 1 to the latter, i.e., the former is regarded as twice as important as the latter. Suppose that the program held the information that if one looks at two-move patterns the player has chosen Scissors after Rock 8 out of 20 times and if one looks at the current three-move pattern (say Tissue, Tissue, Rock) the player has never chosen Scissors after Rock but has chosen Tissue 5 times out of 6. This means that the player has a tendency to follow Rock by Scissors except when he has just played two Tissues, when he has then played another Tissue on every occasion except one. We might expect him to choose another Tissue. Our evaluative function would be:

2*(number of times the player has played x in this two-move position/number of times the player has been in this position) + 1*(number of times the player has played x in this three-move position/number of times the player has been in this position)

If we suppose that Scissors had only been played once after this three-move position this would give for Scissors:

2*(8/20) + 1*(1/6) =0.96

If we use the same function on the number of times Tissue has been played we might find it is six times in the two-move pattern and five in the three-move pattern, giving:

2*(6/20) + 1*(5/6) = 1.43

Thus although the player has more often played Scissors after Rock than Tissue after Rock and the evaluative function gives double the weighting to the two-move pattern, the fact that the three-move pattern shows a clear tendency for Tissue means that the program will on this occasion expect Tissue and therefore choose Scissors as its counter-move.

If this is not complicated enough then you could also add on the history of four-move patterns, and so forth, giving them weightings in the evaluative function which reflected their supposed degree of importance. Obviously the success of the whole program relies on having a sensible evaluative function but for many games this is difficult to work out and can only usually be done by trial and error. First you establish your evaluative function with weightings that you think are reasonable and you RUN the game. Play it a few dozen times and see how easy it is to beat. If it is easy then make a major adjustment to just one of your weightings. (Only alter one at a time or you will not know which is having the observed effect.) Play the same number of games and see if you were able to win more games or fewer. If you won more change the weighting back and try a different alteration. If you won fewer then you are on the right lines and you might want to alter that weighting even more in the same direction. In the above example we might decide that the result was correct but it showed that more weighting should be given to the three-move patterns, so we may change the formula to 3*(two-move pattern)plus 2*(three-move pattern), which increases the relative importance of the latter. Eventually you should get to a point where every alteration you can think of only makes the program play a worse game, in which case you have found the best evaluative function you can using the variables you are giving it. If the program is still playing miserably then you have not chosen the right variables in the first place and no tinkering with weightings will alter the fact that your description of the game is flawed.

5.2 An intelligent opponent

In this section we will begin to look at a new computer board game which has a certain amount of intelligence. What we want is a program which can play the game instead of a real opponent, so the program has to know the rules of the game and have some idea of how to use those rules to win. The program can be thought of as having two parts. The first part is the part that replaces the board and components of the game (such as playing pieces, dice, cards, etc.). Some people are quite happy with programs that simply provide this. For them the micro game is just the same as an ordinary board game except that it is played using a keyboard and TV and both players are human using the computer as a means of playing.

The second part of our program is that which replaces the opponent. This is the part which is, in some sense, intelligent. It 'understands' the game and also has some 'ideas' on how the game should be played. Obviously it would be quite easy to write a program that cheats in any game because all the information about the game, together with all the coding of the rules, is held by the computer. Consequently our two parts should be kept separate. It is unfair for the human player if the computer is able to access information or carry out operations which a human player would be unable to do. For example, it would be a pointless game that allowed the computer to make any moves it liked but restricted the human player solely to the legal moves. If the computer can cheat then there is no point to the game. A game is after all a series of rules which *both* players have to agree to. If one player opts out then the game cannot be played—the rules do not work.

Consequently, we must write the second section of the program as an imitation of a human player. It should be able to do what a human player can do, and no more. If we tried to make it do exactly what a human player does we would have a number of problems. Firstly, the actual psychology of playing games (like many other aspects of human behaviour) is not very well understood and what is understood can be difficult to describe. Secondly, most players play in very complex and usually inefficient ways. A player might look at one move, examine some of its possibilities, then notice another possible move, then look at a third, then go back to the first, then perhaps tentatively move a piece, then change his or her mind, begin looking at a totally different section of the game, and finally shrug and simply play any move to avoid thinking about it any more. Acting in this way is usually an inefficient strategy, wasting time and resources, and generally not producing the best move, but it is characteristic of human play. Not only would it be foolish to program a computer to play an inefficient strategy but it would be wasteful of the computer's resources and, as we shall see, memory is an important feature of intelligent games.

The third reason that it would be silly to write a program that played in the same way as a human being, even if it was possible, is that in some ways computers can play better than human beings. For example, there are few human Chess players alive at the moment who are capable of consistently beating the best Chess programs around. The strategies built into the computer are better than the strategies which human beings typically use. Why are they better? They are superior to human strategies because they can be thorough in searching for the best possible move, they can accurately remember which moves have been looked at and which have shown most promise, and they can explore a number of different options of a high degree of complexity without becoming confused or making mistakes.

All this assumes that the programmer of such a superior program has analysed the game correctly. The reason that the current generation of Chess games is so successful is that a great deal of research and experiment has been carried out on such programs over the last 20 years. It was thought that in learning how to program a machine to play Chess major discoveries would be made about human intelligence and problem-solving because the game was regarded as one of the most difficult of human intellectual activities. Consequently, many academics and programmers devoted a great deal of their time to making such programs as efficient as possible. As it turns out not very much has been learned about human intelligence and relatively little has been learned about making machines intelligent, but a great deal has been learned about playing Chess and about getting a machine to play Chess.

When designing a program which is to act as a human player it is important to decide what kind of player we want the program to be. It is not difficult to program a game which makes its moves randomly, but such a game would be relatively easy to play and easy to beat. To make a game play with a degree of intelligence is quite difficult, but to make it better than the best human player is well-nigh impossible. In other words, we must decide on a level of play which is cost effective in terms of the development time available. One reason that there is, at the time of writing, no commercial version of Go available for a computer is that an enormous number of man-hours would be required to construct a game which could play at the level of only an average human being, so no software firm thinks it is worth the investment. On the other hand, if you are an ardent Go player you may be prepared to spend that amount of time because the game is worth much more to you than simply making money out of it. (However, there are a number of other problems in writing a Go program.)

A second factory normally limits the programmed intelligence of a game, and that is the capacity of the computer. The drawback with comprehensive game search routines is that they use an enormous amount of memory and they can take a long time to execute. On the Electron we are limited to about 24K for a program (although BBC owners have mode 7 which gives about 31K of available RAM). For most purposes this would be plenty, but intelligence can require a very large amount of temporary storage while its routines are being carried out so that every byte becomes precious. We will see, for example, that the game used in this chapter is in some way badly written because of the need to leave enough memory for the intelligent routines.

Similarly, the speed of the processor can be an important factor. The essential principle of getting a computer to think is to make it explore every possible alternative. This means that it has to do a great deal of work, which takes even a computer a great deal of time. One of the advantages of the human method of decision-making is that is is often faster than that of the machine because certain options are ignored from the beginning. Thus if we are writing our routines in BASIC rather than machine code the process of deciding on the move can be intolerably slow. The programmer thus has to decide on what is a tolerable amount of time for such processing, which of course will depend on who is using the program. If it is to be commercially available then it is important that it be as fast as possible, but if it is only for the use of the programmer, a dedicated games player, then perhaps delays of hours can be tolerated between moves.

5.3 Dilemma—the basic game

Before looking at the intelligent aspects of this chapter's game, Dilemma, we will briefly examine its nature. The principle is quite simple, if a little tortuous to describe. A board is set up as an 8 × 8 array of coloured squares. No two squares of the same colour will be adjacent, vertically or horizontally. Each player has three pieces and the first player to move one of his pieces from one side of the board to the other is the winner. Movement is alternate and may be of any piece but must be forward, either directly or diagonally, and only one square may be moved at a time. The only other constraint is that a piece can only move to a square which is the same colour as one directly in front of one of the three pieces of the opponent. Consequently, if all the squares immediately in front of the opponent's pieces are white then it is only possible to move to a white square. If no white square is close enough to one of the pieces the player would have to forfeit a move. It is thus possible to obtain a draw where no piece on either side can be moved. In this case we could decide that the player with a piece nearest the opponent's baseline is the winner.

The set-up part of this program must therefore be to build a board of eight by eight squares with no adjacent squares of the same colours. The starting positions of three pieces for both players must then be decided. The main program must know where each piece is and must be able to print the pieces on the board and move them to the correct locations when desired. It must also be able to tell if a player has won and should be able to print error messages if illegal moves are attempted. In addition, it must also include a mechanism which allows the human player to specify which piece is to be moved and where it is to be moved to.

For the Electron this immediately adds some constraints to our design. In the first place we need a multicoloured board together with two different colours for the opposing pieces. As the game is not very interesting unless the board has at least four different colours this means a minimum of six colours need to be displayed, and this limits us to mode 2. Consequently we only have 12K for the program (20K being used for the graphics), which means that the intelligence routines have to be brief.

Our other main decision is how to input the user's moves. One normal method would be to allow the input of coordinates. This would mean that each time the player wanted to move, a letter and a number (let us say) would have to be typed to indicate the square to be moved from, and another letter and number to indicate the square to move to. If a mistake was made the directions would have to be re-typed. This is a tedious if simple method of input which annoys some players so much that they never bother with the game. A better, quicker, and more direct method is to use the cursor keys to identify the chosen piece and square. Although the user still has to press two keys there is no need to worry about coordinates and no need to read small letters off the screen or to worry about typing mistakes. The main drawback with the method is that it will use more memory than the coordinate approach. Still it is probably preferable to design a less intelligent game that a player enjoys playing than a more intelligent game that he or she does not want to play.

A third method is to let the micro indicate a possible piece to be played. This is used in our version of Dilemma. At each turn the program first checks that at least one move is possible for the player and, if it is, one of the player's pieces is flashed and the player is asked whether that piece is to be moved. If the response is 'N' for 'No' another piece is flashed and the program continues to cycle through all three pieces until the player types in 'Y' for 'Yes'. When 'Y' is typed the player must select one of the three legal moves by typing 'L' for 'Left diagonal', 'R' for 'Right diagonal' or 'F' for 'Forward'. If the player attempts an illegal move, e.g., by trying to move onto a square which is already occupied, the cycle is repeated. In this way the player receives a clear indication of what is required at each stage and the program has to do only a small amount of checking to ensure that the input is allowed. It is difficult for the player to choose an impossible move and impossible for an illegal move to be carried out. Most importantly, the amount of work the player needs to do is kept to a minimum. Only two keypresses are needed in some cases, with a maximum of four on any turn, provided that an illegal move is not attempted.

The program works by holding the current position of all six pieces in the game in an array called current%. The routine needs only to flash the three positions held in array elements 4 to 6 (the player's piece number, because the machine controls pieces 1 to 3) one at a time. This can be done by constantly calling a FOR . . . NEXT loop whose variable moves between 4 and 6, but our version uses a REPEAT . . . UNTIL loop which, rather like a FOR . . . NEXT loop, increments a counter from 4 to 6 while carrying out the necessary tests each cycle. When the variable reaches 6 it is reset to 4. In this way the whole operation is kept within one loop with only one test for the exit, rather than the more complex structure that would be needed to keep using a FOR . . . NEXT loop. Almost certainly if you tried to use a FOR . . . NEXT loop you would need to use a conditional GOTO loop which loops back within the program. While perfectly acceptable this is a habit to avoid if possible because it can lead to code which is difficult to decipher and adapt.

5.4 The program for Dilemma

The complete listing for the game Dilemma is given in Fig. 5.2. As described above it is a micro version of a board game using mode 2 graphics.

```
10   CLEAR
20  MODE 2
30  PROCinit
40  PROCboard
```

Figure 5.2 'Dilemma' (*continues*)

```
 60
 70   REM ****   Main Loop   ****
 80  REPEAT
 90      PROCis_move_poss(1,3,4,6,1)
100      IF check = 0 THEN draw = draw+1 ELSE PROCumove :
         draw = 0
110       PROCwin
120      IF win>0 THEN 160
130      PROCis_move_poss(4,6,1,3,-1)
140      IF check = 0 THEN draw = draw+1 ELSE
         PROCmachinemove: draw = 0
150      PROCwin
160         UNTIL draw>1 OR win >0
170  IF draw>1 THEN PRINT "A Draw " ELSE PRINT name$(win);
     " wins"
180  END
190
200
210  DEF PROCboard
220  FOR X% = 0 TO 63
230      ?(piece%+X%) = 0
240      NEXT
250  FOR X% = 1 TO 3
260      R% = RND(8)-1
270      IF ?(piece%+R%) <>0 THEN 260
280      ?(piece%+R%) = X%
290      current%(X%) = R%
300      NEXT
310  FOR X% = 4 TO 6
320      R% = RND(8)-1
330      IF ?(piece%+R%+56) <>0 THEN 320
340      ?(piece%+R%+56) = X%
350      current%(X%) = R%+56
360      NEXT
370  FOR Y% = 0 TO 7
380      FOR X% = 0 TO 7
390          R% = RND(6)+1
400          IF board%?(X%-1+(Y%*8)) = R% OR
             board%?(X%+((Y%-1)*8)) = R% THEN 390
410          board%?(X%+(Y%*8)) = R%
420          PROCsquare(X%,Y%)
430          NEXT
440      NEXT
450  GCOL 0,0
460  FOR X% = 160 TO 832 STEP 64
470      MOVE X%,0
480      DRAW X%,640
490      MOVE 0,X%+32
500      DRAW 960,X%+32
510      NEXT
520  ENDPROC
530
540
550  DEF PROCsquare(X%,Y%)
560  VDU 5
570  MOVE 300+(X%*64),640-(Y%*64)
580  GCOL 0,?(board%+X%+(Y%*8))
590  PRINT sq$;
600  PROCpiece
```

```
 610  VDU 4
 620  ENDPROC
 630
 640
 650  DEF PROCpiece
 660  IF ?(piece%+X%+Y%*8)>3 THEN pcol = 0 ELSE pcol = 1
 670  IF ?(piece%+X%+(Y%*8))>0 THEN GCOL 0,pcol :
      PLOT 0,-64,16 : PRINT pc$ : GCOL 0,R%
 680  ENDPROC
 690
 700
 710  DEF PROCinit
 720    win = 0
 730  DIM board% 64
 740  DIM piece% 64
 750  DIM current%(6)
 760  DIM name$(2)
 770  name$(1) = "The computer "
 780   name$(2) = "The player "
 790  q% = FALSE
 800  VDU 28,0,5,19,0
 810  VDU 24,0;0;1023;800;
 820  VDU 23,241,&18,&3C,&7E,&3C,&18,&18,&3C,&7E
 830  VDU 23,242,&00,&00,&00,&00,&18,&3C,&7E,&00
 840  VDU 23,240,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
 850  sq$ = CHR$240+ CHR$8+ CHR$10+ CHR$240
 860  pc$ = CHR$241
 870  DIM p%(3),q%(3)
 880  ENDPROC
 890
 900
 910  DEF PROCumove
 920  x = 3
 930  REPEAT
 940      REPEAT
 950          x = x+1
 960          IF x>6 THEN x = 4
 970          PROCflash(x)
 980          CLS
 990          PRINT "Move this piece?"
1000          REPEAT
1010              yn$ = GET$
1020              g = INSTR("YyNn",yn$)
1030              UNTIL g<>0
1040          CLS
1050          PROCflash(x)
1060          UNTIL g<3
1070      REPEAT
1080           PRINT "Move Left, Forward     or Right?"
1090          flag = 0
1100          g$ = GET$
1110          g% = INSTR("LlFfRr",g$)
1120          IF (g%<3 AND current%(x) MOD 8 = 0) OR
              (g%>4 AND (current%(x)+1) MOD 8 = 0) THEN PRINT
              "Impossible" : flag = 1
1130          UNTIL g% <>0 AND flag = 0
1140      sq = current%(x)-8+( INT((g%+1)/2)-2)
1150      colsq = ?(board%+sq)
1160      PROCchecksqok(1,3,1)
```

Figure 5.2 (*continues*)

```
1170  IF check = 0 THEN PRINT "impossible"
1180  UNTIL check = 1
1190 PROCupdate(x,0)
1200 ENDPROC
1210
1220
1230 DEF PROCflash(x)
1240 VDU 5
1250 GCOL 3,8
1260 MOVE 300+( FNcurx(x) *64),624-(FNcury(x)*64)
1270 PRINT pc$
1280 VDU 4
1290 ENDPROC
1300
1310
1320 DEF PROCchecksqok(os,oe,param)
1330 check = 0
1340 FOR i = os TO oe
1350 p =?(board%+current%(i)+(8*param))
1360 IF p = colsq THEN check = 1
1370 NEXT
1380 FOR i = 1 TO 6
1390 IF current%(i) = sq THEN check =0
1400 NEXT
1410 ENDPROC
1420
1430
1440 DEF PROCupdate(piece,pcol)
1450 VDU 5
1460 MOVE 300+(FNcurx(piece)*64),624-( FNcury(piece)*64)
1470 GCOL 0,?(board%+current%(piece))
1480 PRINT pc$
1490 ?(piece%+current%(piece))=0
1500 current%(piece) = sq
1510 ?(piece%+sq) = piece
1520 MOVE 300+(FNcurx(piece)*64),624-( FNcury(piece)*64)
1530 GCOL 0,pcol
1540 PRINT pc$
1550 VDU 4
1560 ENDPROC
1570
1580 DEF FNcurx(a) = current%(a) MOD 8
1590 DEF FNcury(a) = current%(a) DIV 8
1600
1610
1620 DEF PROCmachinemove
1630 CLS
1640 PRINT "I´m thinking...."
1650 PROCchoose
1660 PRINT "Aha!"
1670 IF r>0 AND s>-2 THEN PROCupdate(r,1) ELSE
     PRINT "No move possible"
1680 ENDPROC
1690
1700
1710 DEF PROCchoose
1720 bestmac = -900
1730 bestbest = -900
1740 bestmove = -900
```

```
1750   memmove = -2
1760 mempiece = 0
1770 maccount = 0
1780  defaultmove = 0
1790 defaultpiece = 0
1800  FOR piece = 1 TO 3
1810      FOR move = -1 TO 1
1820          mem2 = -2
1830          check = 0
1840          sq = current%(piece)+move+8
1850        colsq = ?(board%+sq)
1860        REM Don´t go off the edge
1870        IF (move = -1 AND ( FNcurx(piece) MOD 8 = 0))
            OR (move = 1 AND (( FNcurx(piece)+1) MOD 8 = 0))
            THEN check = 0 ELSE PROCchecksqok(4,6,-1)
1880        IF check = 1 THEN PROCcount(piece,move) :
            defaultmove = move : defaultpiece = piece
1890        REM Record best move for thi piece
1900        IF bestmac > bestbest AND check = 1 THEN
            bestbest = bestmac : mem2 = move
1910        NEXT
1920      bestbest = bestbest+( FNcury(piece))/2
1930      IF check = 1 AND bestbest > bestmove THEN PROCchosen
1940      NEXT piece
1950   IF mempiece<>0 AND memmove>-2 THEN
       PROCshow ELSE PROCdefault
1960  sq = current%(r)+8+s
1970  colsq = ?(board%+sq)
1980  ENDPROC
1990
2000
2010  DEF PROCis_move_poss(os,oe,ps,pe,param)
2020  check = 0
2030  FOR k = os TO oe
2040       FOR i = ps TO pe
2050           FOR j = -1 TO 1
2060              IF j+param = 0 AND param = 1
                  AND (current%(k)) MOD 8 = 0
                  THEN 2110
2070              IF j+param = 0 AND param = -1
                  AND (current%(k)+1) MOD 8 = 0
                  THEN 2110
2080              IF j+param = -2 AND (current%(k)) MOD 8 = 0
                  THEN 2110
2090              IF j+param = 2 AND (current%(k)+1) MOD 8 = 0
                  THEN 2110
2100               IF ?(board%+current%(k)+(8*param)) =
                   ?(board%+current%(i)-(8*param)+j) AND
                   ?(piece%+current%(i)-(8*param)+j) = 0
                   THEN check = 1
2110              NEXT
2120          NEXT
2130      NEXT
2140   IF check = 0 THEN PRINT "No move possible"´;"Any key to
continue" : h = GET
2150  ENDPROC
2160
2170
2180  DEF PROCwin
```

Figure 5.2 (*continues*)

```
2190  FOR i = 1 TO 3
2200      IF current%(i) >55 THEN win = 1
2210      IF current%(i+3) <8 THEN win = 2
2220      NEXT
2230  ENDPROC
2240
2250
2260   DEF PROCcount(p,m)
2270   REM Hold three temporary positions in p%(x)
2280  FOR i = 1 TO 3
2290      p%(i) = current%(i)
2300      NEXT
2310  p%(p) = sq
2320  oppcount = 0
2330  bestmac = 0
2340   REM Count all possible opponent moves
2350   FOR j = 4 TO 6
2360     FOR opmove = -1 TO 1
2370       sq = current%(j)+opmove-8
2380       colsq = ?(board%+sq)
2390       PROCchecktwo(1,3,1)
2400       IF ck = 1 THEN oppcount = oppcount+(9-(sq DIV 8)) :

           maccount = 0 : PROCmaccalc
2410       NEXT opmove
2420      IF bestmac<maccount THEN bestmac = maccount
2430      NEXT j
2440  PROCinfront
2450  oppcount = oppcount*forward
2460  bestmac = (bestmac*2)-oppcount
2470  ENDPROC
2480
2490
2500    DEF PROCchecktwo(os,oe,param)
2510  ck = 0
2520  FOR i = os TO oe
2530      p = ?(board%+p%(i)+(8*param))
2540      IF p = colsq THEN ck = 1
2550      NEXT
2560  FOR i = 1 TO 6
2570          IF current%(i) = sq THEN ck = 0
2580      NEXT
2590  ENDPROC
2600
2610
2620  DEF PROCmaccalc
2630  FOR k = 1 TO 3
2640      q%(k) = current%(k+3)
2650      NEXT
2660   q%(j-3) = sq
2670  FOR k = 1 TO 3
2680      FOR kmove = -1 TO 1
2690               colsq = ?(board%+p%(k)+kmove)
2700          PROCcheckthree(4,6,-1)
2710          IF ch = 1 THEN maccount = maccount+1
2720          NEXT kmove
2730      NEXT k
2740  ENDPROC
2750
```

```
2760
2770  DEF PROCcheckthree(oe,os,param)
2780  ch = 0
2790  FOR i = os TO oe
2800         p = ?(board%+q%(i-3)+(8*param))
2810      IF p = colsq THEN ch = 1
2820      NEXT
2830  FOR i = 1 TO 6
2840      IF current%(i) = sq THEN ch = 0
2850      NEXT
2860  ENDPROC
2870
2880
2890  DEF PROCinfront
2900  forward = 0
2910   FOR z = 4 TO 6
2920      IF FNcury(z)>forward THEN forward = FNcury(z)
2930      NEXT
2940  ENDPROC
2950
2960
2970  DEF PROCchosen
2980  bestmove = bestbest
2990  mempiece = piece
3000  memmove = mem2
3010  REM best piece and best move
3020  ENDPROC
3030
3040
3050  DEF PROCshow
3060  r = mempiece
3070  s = memmove
3080  ENDPROC
3090
3100
3110  DEF PROCdefault
3120  r = defaultpiece
3130  s = defaultmove
3140  ENDPROC
```

Figure 5.2 'Dilemma' (*continues*)

As in Pantry Panic the graphics of Dilemma are user-defined graphics. Mode 2 is used to get the range of required colours so the graphics are rather 'chunky'. They are also rather wider than they are tall, so to produce the coloured squares for the board two smaller rectangles are drawn. A string of characters is created called sq$ and each time we wish to print a square that string is printed. This is more elegant than repeatedly calling a 'square drawing' routine. The string is defined in line 790 and is made of two of the user-defined rectangles (CHR$ 240, defined in line 780) plus cursor movement characters (CHR$ 8 and CHR$ 10, to move the cursor to the correct position for printing the second rectangle). The other user-defined graphic, CHR$ 241, is the shape for the player's pieces. This is called pc$ for ease of reference and use but there is no reason why it could not be called using VDU 241 or PRINT CHR$ (241) instead.

PROCboard is the main routine which controls the graphic display. Firstly, it places the three pieces of the player and the three of the computer in random places in the byte array that represents the board, piece%, and puts the current position of each piece in an integer array called current%. A byte array differs from both the integer and numeric arrays previously discussed when describing Pantry Panic. Instead of being a multidimensional matrix of elements, into each of which a number can be placed, a byte array is simply a block of memory of specified size to which the first byte is given a name. So in line 740 "DIM piece% 64" means 'set aside a block of 64 bytes starting with the byte we will call piece%'. It is not necessary for the programmer or the user to know the address of piece%, as long as the micro knows it. We can then write bytes into and read them out of this block by referring to "piece% + n", where n is the number of the byte we want to refer to. The main advantage of a byte array is that it uses only one byte per number, provided the numbers are between 0 and 255, as compared with the ordinary numeric array which uses four bytes per number. For our purposes the 64 bytes represent the 64 squares of the board for the game. Just as in Pantry Panic we used a map of the screen to hold the information so in Dilemma we will use one block of bytes to hold the current positions of the pieces and another to hold the colours of the squares on the board.

PROCinit then fills this second byte array called board% with a series of random numbers in the range 2 to 7. These are the colours of the squares. Colours 0 and 1 are reserved for the pieces. If you wished to change these colours any colours could be assigned to these logical values using VDU 19, but I have kept the default values. As the random squares are chosen line 370 ensures that no two squares of the same colour are adjacent horizontally or vertically.

Having chosen the colour of a square the routine then PRINTs it. This is why the routine is so slow. It could be accelerated if the colours were all chosen first and then all printed rather than one at a time, but it is not important as the routine is only used once in each game. PROCsquare is the procedure which prints the square. It simply changes the current foreground colour, using GCOL, to that which has randomly been chosen by looking at the value in board% and prints sq$ in that colour. However, it also checks that a piece has not been placed in that square using PROCpiece which checks piece%. If there is a piece in that square then GCOL has to be changed to the piece colour and then pc$ is printed. This means that the foreground colour also has to be changed back when the procedure has finished. Note that each piece is only one graphic character but each square is two. PROCpiece therefore moves the piece into the middle of the square before PRINTing it. This gives a much more satisfactory appearance than a single-character square or an uneven border around the piece. Note also the use of VDU4 and VDU5 to tie the text and graphics cursors together because we are PRINTing a text character (i.e., a user-defined graphic) at a graphics coordinate (using MOVE). A final routine draws a black grid around the squares to make the whole thing look a little neater.

When the board has been set up the game consists of a simple repeated sequence, represented by the main loop of lines 60 to 150. The program first checks that the current player is able to make at least one move. If not, then the variable 'draw' is incremented by one and play passes to the other player. If both players cannot play, i.e., if the variable draw has exceeded 1, then the game is a draw (tested for in line 140). Only three procedures are called by the main loop. PROCumove controls the player's move. PROCmachinemove controls the computer's move and PROCwin tests to see if either player has reached the opponent's baseline.

PROCumove has been described above. It consists of requests for input, a procedure called PROCflash to indicate which piece is being referred to, and various tests to ensure that the desired move is possible and legal. Line 1500 ensures that the player does not try to move a piece off the edge of the board. Because our byte array representing the board is actually a sequence of 64 values and not a matrix of eight squares by eight it is not actually possible to go off the edge of the board. Instead, the piece would appear on the opposite side of the board one rank forward, i.e., eight bytes further into the array. Although this is not allowed in the original game this variation could actually make the basis of an interesting game. If such moves were allowed the game board would be topologically 'cylindrical' rather than 'flat', i.e., it would have the spatial properties of a cylinder. You may like to explore the idea of a board without edges in designing your own game. This is one area where a game can be played on a micro that could not be played in any other way.

The other main checking procedure is PROCchecksqok. This looks at the squares immediately in front of the opponent's pieces to see if the square the player intends to move to is of the same colour and hence legal. When a legal move is attempted PROCupdate is called. This calculates the current position on the screen of the selected piece and then looks at board% for the colour of the square. It then PRINTs pc$ on that square in the square's colour which is, of course, equivalent to blanking out the piece. It then 'moves' the chosen piece by printing it onto the intended square and transferring the values in current% and piece% from the old positions, which are given zero, to the new. This occurs so quickly that it looks as if the piece actually moves across the board. You will realize that this is exactly the same process as used in Pantry Panic to move the figures around the screen.

The rest of the program is devoted to helping the computer make its own move. As this is the key purpose of this program you might like to experiment with various approaches yourself. Because of the modular construction this is quite easy to arrange. PROCmachinemove consists of two lines. Firstly, PROCchoose is called and this returns two values, r which is the piece number to be moved and s which is the square to be moved to. Then PROCupdate is called to move the piece in exactly the same way as for the human player's piece. Consequently, we can write any routines we like to determine the program's move, provided that the routine produces a value for r and a value for s.

The easiest way to do this is to select the values randomly. A routine which can be used in Fig. 5.2 instead of lines 1710 to 1980 and 2260 to 3140 is given in Fig. 5.3. You can see that it is rather shorter than the 'intelligent' version. It consists of a PROCchoose which selects a random value for s and a random value for r, checks that they are legal in the same way as the human moves are checked, and then passes those values back. You will see that the same routine, PROCchecksqok, is used to do this. It simply has different parameters. The first two parameters, os and oe, are 'opponent's start piece' and 'opponent's end piece' and the third parameter, param, holds the direction to examine. If PROCchecksqok is checking squares in front of the human's pieces then it needs to look at the value eight less than the piece position, i.e., 8 * −1, whereas to examine a square in front of a computer's piece it must look at a value eight greater than the position of the piece, i.e., 8 *1. So in the former case param has a value of minus one and in the latter of plus one.

A random routine like this works reasonably well and on one or two occasions it can beat a human player if the squares are arranged in a fortuitous combination. However, most human players do not find it too difficult to beat such a random routine most of the time. Nor is it very interesting to play because it is almost impossible for a player to decide on the routine's most probable course of action and thus to take appropriate steps. In other words the game feels like a random game to the player who is not able to develop any consistent strategy other than to ignore the opponent. Consequently if we want a game like Dilemma to be interesting we must build in at least some intelligence.

```
1530 DEFPROCchoose

1540 REPEAT

1550     REPEAT

1559       REM Choose piece

1560       r=RND(3)

1569       REM Choose move

1570       s=RND(3)

1580       s=2-s

1599       REM Make sure the choice is on the board

1600     UNTIL NOT (s=-1 AND (FNcurx(r) MOD 8=0))
           AND NOT (s=1 AND ((FNcurx(r)+1) MOD 8=0))

1609   REM Make sure the colour is allowed

1610   sq=current%(r)+8+s

1620   colsq=?(board%+sq)

1630   PROCchecksqok(4,6,-1)

1640 UNTIL check=1

1650 ENDPROC
```

Figure 5.3 Random machine movement

5.5 Intelligent play

What would make the program play better? Let us approach this by considering what a human player would be thinking about when trying to win this game. This is one of the minor benefits of attempting to program intelligence—you always have a model to hand. (It can also be a drawback because you are limited to your own knowledge, abilities, and strategic skill in designing your algorithm for the program.)

Obviously a human player would be doing two things. He or she would be trying to win the game and trying to prevent the opponent winning. This means that two types of strategy will be kept in mind at the same time and two considerations at least come into every decision:

1. If I make move x will it increase my chances of winning?
2. If I make move x will I decrease my opponent's chances of winning?

This, of course, means that some kind of evaluation is going on concerning each move, such as:

3. On balance, is the advantage I gain from move x coupled with the disadvantage my opponent gains better than the combination of advantages and disadvantages from move y?

So our player, Igor, is looking at each move for advantages to both sides and then trying to weight each combination of advantages and disadvantages to find the move with the best combination. In other words, to use the jargon introduced in the Rock, Scissors, Tissue example, Igor is assigning a value to each move and comparing them by using some kind of evaluative function. So all we need to do to get our program to play like Igor is to use the same evaluative function and assign the values correctly to each possible move.

It should be clear by now why Dilemma has been chosen as the example of game search intelligence rather than, say, Chess or Backgammon. In Dilemma each player can only move three pieces and each piece can only move one of a maximum of three squares in any one move, making a maximum set of nine possible moves to evaluate each turn. In Chess we might have all 16 pieces faced with four of five moves each, giving 80 or more moves to evaluate. Furthermore, the moves in Dilemma are all of the same kind; we can use the same function to evaluate all the possible moves. However, in Chess this is not the case and functions may be much more complicated as a result.

Actually this need not be the case because the evaluative function can be used not to assess each move as such but the position which results at the end of each move, by asking questions like: Is my king likely to be placed in check? Is the opponent's king in check? Does the opponent have fewer pieces? etc. What kinds of characteristics of a game position will Igor be looking at? The most important thing in Dilemma is to move your pieces forward, especially if the chosen piece is near the opposing baseline. On the other hand you should not make a move which allows the opponent to move a piece onto your own baseline.

The second important aspect is to ensure that you have as many potential moves available as possible. The fewer moves you can make, the easier it will be for your opponent to prevent you from moving. Conversely, you want to reduce the number of moves available to your opponent. Let us take this as the basis of our evaluation and see what can be done with it. We will look at an evaluative procedure usually known as 'minimaxing'. It is called this because it involves trying to find the maximum possible value for your own move which ensures the minimum possible score for your opponent. At first sight this might seem simple but let us examine the idea in more detail.

Suppose we had a game in which only three moves are possible each turn, A, B and C. Let us say that the evaluation score for a move is u (for us) and t (for them). We will not worry about the evaluative function for the moment. Just suppose it to be a reasonable one. So if we make move A the value will be Au, if we make move B the value will be Bu, and if we make move C it will be Cu. Then a simple calculation would tell us which move is best to make. Whichever is the greater of Au, Bu, and Cu indicates the best move. If Au is greater then we should make move A.

However, remember that after we have moved the opponent will also move. He or she will also have three choices: let us say X with value Xt, Y with value Yt, and Z with value Zt. Whichever move we make from A, B, or C the opponent will have three moves. We can represent this by saying that in two turns (after we have had a turn and the opponent has had a turn) there will be nine possible outcomes, as follows:

Au – A (Xt)
Au – A (Yt)
Au – A (Zt)
Bu – B (Xt)
Bu – B (Yt)
Bu – B (Zt)
Cu – C (Xt)
Cu – C (Yt)
Cu – C (Zt)

That is to say, the outcome of any move will be its value to us minus the value to the opponent, and there are nine such possible outcomes. Of course the opponent is going to choose the outcome which is the best of those nine, so there are really only three likely outcomes—the best choice for the opponent following move A, the best choice for the opponent following move B, and the best choice for the opponent following move C.

Now suppose the value for move A was 7, for B was 4, and for C was 6. If we were using just a one-ply search, i.e., if we were just looking at one turn, we would choose A. But suppose the best move that the opponent can make after A gives a score of 9, the best after B a score of 2, and the best after C a score of 5. Which is now the best move—A, B, or C? Clearly B is our best move because it results in a net gain of 2. Look at the scores:

Move	Ply 1	Ply 2	Net value
A	7	−9	−2
B	4	−2	+2
C	6	−5	+1

If we represent the value to us as a positive score and the value to the opponent as a negative score (because a positive score for the opponent reduces our chances) then the net benefit of any move will be the sum of our score and the opponent's best reply. B is, at one ply, our worst move, but because it also results in the worst outcome for the opponent it is actually the best move when the possible replies are taken into consideration.

Now if we were being really thorough we would want to go further than this. Suppose we made move B because of our two-ply analysis and then found that our best score for the next move was only 1, whereas the worst the opponent could get on the next move was 12 (i.e., −12 from our point of view). This would be a four-ply analysis and the overall value of move B, the move that starts the sequence off, would be $4 - 2 + 1 - 12 = -9$ (assuming that both players make the best choice each turn), which does not look very good. However, what about the next ply down?

You will see that an ideal analysis would follow all the stages of evaluation from the first move to the end of the game. You will probably also see that this would be impossible to do for most games on a microcomputer. Staying with our three-move game, suppose a complete game could be played in eight turns, i.e., an eight-ply search would guarantee to give the best first move. (It cannot, of course, because there are eight squares to traverse so the smallest full search would be 15 ply.) For an eight-ply search we would need to evaluate 3*3*3*3*3*3*3*3 positions to find the best overall score, i.e., a total of 6561 calculations. This might take some time. More importantly, at each ply at each level of evaluation we would have to hold all the currently possible positions because we would not know which ones to discard until we have come to level eight. So we might have to hold over 6000 game positions in memory at one time. Even if our game had a minimal nine squares we would presumably need nine bytes to hold each possible position, so we would need 6561*9 bytes to remember every position, which is roughly 60K of memory. And this is supposedly a simple and trivial game!

Of course I am exaggerating here. There are techniques which can reduce the storage needed for such an evaluation quite dramatically. Our nine-square board can be held in a bit map of nine bits (one byte plus one bit), and we can throw away many positions as we work through the search and evaluation as they prove much worse than any current position. Even so, for most practical purposes searches at levels below three ply can become difficult to code, and this is certainly the case on the BBC and Electron.

Our routine only looks at two ply. It considers each possible move in turn and counts the number of moves that would be available to the opponent. For each of those moves it counts the number of moves that could result. PROCchoose selects each move in turn and calls PROCcount if a possible move is found. PROCcount counts all the possible opponent's moves and calls PROCmaccalc if a legal move is found. PROCmaccalc counts the number of moves that would then be available to the machine as a result. These values are compared with the best values found so far and, if they are better, replace them. Finally, the variable 'bestbest' is calculated and the move which results in bestbest (the best score of all the possible best scores) is remembered in memmove and mempiece. These values are passed back to r and s and this is the move that is made, using PROCupdate as before.

To do all this it is necessary to hold the positions of the pieces that would result at each level of analysis so that the positions can be evaluated. Consequently most of the programming (and memory) taken up by the 'intelligent' portion of this program is simply in remembering the current projected board set up for evaluation. PROCchoose sets the variables to very low values so that the evaluation will be higher. It is no good setting them to zero because on some occasions the best move may still have a negative evaluation. PROCcount holds the position of the opponent's pieces that would result if a possible move is made. PROCchecktwo counts the number of legal moves the opponent could make from that current position, held in p%(). PROCmaccalc then holds the position that would result from each of the opponent's legal moves for that position in q%() and PROCcheckthree counts the number of legal moves that the computer could make as a result of that position. This uses a fair amount of memory and programming. The program could be made more elegant by using recursion, but as recursive routines are notoriously difficult to read and understand I have tried to keep things relatively simple. (Now you know what simplicity is in programming!)

On some occasions no evaluation may provide a sensible move, in which case PROCdefault is called. Two variables, defaultmove and defaultpiece, have been given the values of an arbitrary move which the routine has found to be legal (in line 1880). These values are returned as s and r if no good moves are found by the evaluation.

The evaluation function itself is split into several parts. This was a result of trial and error experimentation with the program. It was found that the evaluation as just outlined, i.e., entirely based on maximizing the computer's moves and minimizing the player's moves, led to some odd moves. For example, the nearer the opponent's pieces get to the far edge the fewer moves are available to the pieces, so the program tended to ignore the most dangerous of the player's pieces. It would also sometimes not move on to its own final rank and thus win the game because another of its possible moves reduced the opponent's available moves by a large amount. Consequently, some weighting has to be added to ensure that it regards the pieces nearest the destination edges as the most dangerous.

This is done partly by PROCinfront. PROCinfront finds out the rank of the frontmost piece of the player and returns this in the variable called 'forward'. Forward is used in line 2450 to weigh the calculation so that oppcount (the number of moves available to the player as a result of the currently evaluated move) is multiplied by the rank of the frontmost piece. The effect of this is that as the player's pieces get nearer and nearer to the edge the number of moves available to the player becomes more and more significant to the evaluation. In the early stages it is almost ignored but in the final stages it becomes the most important factor.

The second stage of the evaluation is in line 2460. This performs the evaluation not simply by subtracting the opponent's possible moves from the computer's but by doubling the importance of the computer's moves. In this way the program will favour attacking rather than defensive moves, i.e., it will tend to make the move that leads it forward rather than the one that reduces the player's possible moves, all other things being equal. However, as 'forward' increases so this weighting becomes overruled.

The final stage of the evaluation is line 1920. Here the current position of the computer's piece that is being evaluated is added to the equation. The rank that the piece is on is divided by two and added to the evaluation. Consequently, the closer it gets to its destination edge the higher will be the added factor, and if two pieces would result in the same overall position for both sides then the one nearer the destination will be chosen.

You might like to play around with the values in each of these three lines to see what happens to the computer's play. Defensive play will be encourged by reducing the weighting in line 2460 but more draws will result as fewer and fewer moves become available. Increasing the weighting in line 1920 will result in the frontmost piece of the computer being given more and more of the play, which is probably not a good strategy because if it gets blocked then a piece from the rear has to do all the running. Tinkering with 'forward' will make the most advanced player's piece more or less significant. As it stands the program plays reasonably well and gives an average player something to think about, but there are always positions in which it makes mistakes and you may find a better balance of weightings.

6 ADVENTURE BASICS

6.1 Adventure structure

Let us now begin to look at one of the most popular kinds of micro game where the scope for intelligent programming is almost unlimited—the adventure game. In this chapter we will consider some of the basic considerations for adventure design while in Chapter 9 we will build a complete adventure which has some 'intelligent' routines.

Let us first look at the story structure of adventure games. The story is what gives sense to the game, making it into something which is a coherent pattern rather than a random series of events. Planning an original story, or what games designers generally call a 'scenario', can be the difference between a game which feels like a computer program and a game which feels like a world worth exploring.

A story is a series of events leading up to a major consequence. The events happen to a major character or group of characters and the consequence is often either the character's death or the achievement of a particular objective. In game terms the character is the player's 'piece' or persona (discussed in the next section) and the consequence is either successful completion of the game (= achieving the objective and giving the maximum reward) or unsuccessful completion (= death).

An event in a story usually fits the following formula:

MAIN CHARACTER + PLACE + TIME + OTHER CHARACTERS + OBJECT(S) + POSSIBLE ACTIONS + POSSIBLE CONSEQUENCES

So an event in an adventure game should ideally have all the components in that formula, and any adventure program is a system for producing sets of such components, each set forming a coherent event. Not every event will need all the components but some will always be needed. A series of such events forms the story and this corresponds to the player character's progress through the mapped locations in the adventure game.

The links between mapped locations may be totally random or totally structured or a combination of both, and similarly with events. The advantage of random determination is that it is easy to program and can give great variety. Its main disadvantage is that a random game rapidly becomes boring because it consists of a series of unpredictable events with no logical relationship, i.e., a series of purely local rewards. A decision made in one location will have no effect on subsequent actions unless the player's character is altered in some way in that location—becoming weaker, perhaps, or finding a laser sword. Adding this kind of alteration to a basically random game is quite easy to do. It provides a simple way of adding some structure to a game. If no such alteration occurs in any location the events are unconnected and we really have a series of small games rather than one large one.

This approach—using a random series of events which connect only by their effect on the central character—is typically a series of rooms or caves in each of which there is a percentage chance of OTHER CHARACTERS (usually monsters) or OBJECTS (treasures, weapons, food) or ACTIONS (falling into a pit, becoming ill, reading an inscription) occurring. For example each room in a dungeon could contain an event constructed in the following manner:

```
1. GENERATE RANDOM NUMBER A IN THE RANGE 1–10
2. GENERATE RANDOM NUMBER B IN THE RANGE 1–10
3. GENERATE RANDOM NUMBER C IN THE RANGE 1–10
4. IF A IS GREATER THAN 4 THEN GENERATE RANDOM
   NUMBER D IN THE RANGE 1–4
5. THIS ROOM CONTAINS MONSTER (D)
6. IF B IS GREATER THAN 4 THEN GENERATE RANDOM
   NUMBER E IN THE RANGE 1–4
7. THIS ROOM CONTAINS OBJECT (E)
8. IF C IS GREATER THAN 6 THEN GENERATE A RANDOM
   NUMBER F IN THE RANGE 1–4
9. THIS ROOM CONTAINS ACTION (F)
```

Random numbers A, B, and C decide whether there should be a monster, object, and action in a particular room; random numbers D, E, and F select the appropriate monster, object, and action if there is one. These would be set up in the program using arrays or subroutines from which random items could be called. Each randomly chosen item would then be a set of variables which potentially modifies the character in some way. Sample monsters, objects, and actions are outlined in other chapters. This sort of game is less common than it used to be because designers have developed various more rewarding approaches, including the addition of some intelligence to originally random games.

The opposite approach, that of a totally structured series of events, is typical of the puzzle-type adventure game. Here the task is not to survive as many randomly generated events as possible, but to discover the puzzle or story and pass through each of the planned events in the correct order. For example, Igor may find that he cannot progress in the game until he has discovered how to open a rusty trapdoor. To discover this, he must bribe a goblin, which means he has to obtain some money. To get the money he must first get past the headless ghost—and so on. The advantage of this type of game is that it is a real test of the player's abilities, intelligence, logical power, and imagination, and is not merely a test of reactions or responses to a random series of accidents. The puzzle adventure game has been likened to the crossword—it demands the same class of skills, including language skills, and all parts must be solved to complete the whole. Therefore if it is well written it demands intelligence in the player. As players become more sophisticated and clever so adventure games must develop equivalent intelligence in order to keep ahead of the players and provide new and exciting forms of play.

The main disadvantage of a totally structured series of events is that the game is the same each time it is played. There is none of the novelty or unpredictability of a random dungeon, the initial stages may become tedious with repetitive play, and the game once solved will never be played again. It is also a much more severe test of the programmer's imagination and ability, as a fully structured game demands a highly structured program.

When designing our game we must bear in mind the fact that players should feel in part at least as if they are progressing through a story. It does not matter if some of the elements are random or fully predetermined, but they should seem coherent from Igor's point of view. It is important, therefore, that his character makes sense.

6.2 The player character

The character is the player's persona. It is the unit that represents Igor in the game. When it is destroyed, the game is over. We will use the word 'character' even when it is a monster or spaceship or vehicle that the player is pretending to be because essentially it is the abilities and behaviour of that 'playing piece' that make combat adventure games entertaining. In the puzzle game, however, the player seldom has a defined or variable character, which is one of its drawbacks. It would be better if puzzle adventure games were designed so that different character configurations were able to approach the solution differently, but this would cause enormous programming problems, as you will see later. In the puzzle adventure game, therefore, it is the player's own personality that is being tested and not the surrogate personality of the character.

What the character is in game or programming terms is a collection of numbers that are altered as the game goes on. For example, if the character is a medieval knight, he might be regarded as:

Speed 4
Defence 5
Attack 4

where the maximum is 6. If he lost his horse or his armour or his sword these numbers might decrease. If he drank a magic potion or found a mace or rode a dragon, they might increase. Every time the player has to make a decision in the game the chance of success will depend on the current value of one or more of three numbers. Therefore, if the player had to decide whether to enter a race, success would depend on speed, but if the decision was about whether to fight a giant, success might depend on both attack and defence.

In other words a character is a collection of variables. Those variables may or may not be related, but to increase the interest of a game it is often a good idea to link such variables in some way. This gives Igor more to think about. For example, mounting a dragon might increase speed but could decrease attack (because the knight's sword cannot reach the enemy from the dragon's back). However, if he gets a lance, this would increase attack value on dragon-back, but might reduce defence because it is more difficult to use a shield with a lance than with a sword. Now, Igor, are you going to get on that dragon or not?

Normally a collection of variables is best kept as an array, so let us start to build up the array for our first hypothetical character. We will call him Sir Jon (his mother wanted him to be a doctor). We will hold his variables in array A(4) and give him four variables to start with—strength, skill, constitution, and knowledge—so we need to DIMension an array with just four variables. This might be wasteful on memory, but when we develop Sir Jon later on we will need the flexibility of an array. Having DIMensioned the array we can READ into it the initial DATA, i.e., the values the variables are initially set to, the abilities that Sir Jon starts off with. Let us make strength and knowledge 2 and skill and constitution 1. So our routine to set up the character would be:

```
8000 REM Set up Sir Jon
8101 DIM A(4)
8929 FOR I=1 TO 4
8030 READ A(I)
8040 NEXT I
9000 REM Sir Jon's data
9010 DATA 2,2,1,1
```

Naturally these values will not be arbitrary. We must have some idea of the likely range of values. Designing a character is tied up to a large extent with what the character is going to do in the game and what the game might do to him. For example, if we wanted the possibility of a weak character fighting strong monsters we might allow a range of 0 to 9 for strength, but not allow any character to be greater than 6.

However, if strength, A(1), is a variable to be used in routines other than the combat routine (such as a routine for lifting heavy objects) we must ensure that the range is also suitable for these routines and that the initial value is set to a meaningful level. Usually we will want the character's variables set to the lowest in the range (if the game is primarily concerned with improving abilities) or to the highest value (if it is a game about avoiding weakness). We might also want to set some variables at mid-point, meaning 'normal' or 'average', if they are the kind of variables that could get better or worse. Let us assume that Sir Jon is average in strength and knowledge, starts off with little training and hence low skill, and being rather undernourished is weak in constitution. If the range for all the variables is 0 to 9 Sir Jon's values could be:

```
A (1) = 4
A (2) = 4
A (3) = 0
A (4) = 1
```

We are expecting him to improve his skill and constitution through discipline and hard work and are also assuming that he could get stronger or weaker, more knowledgeable, or less knowledgeable.

Let us also add a variable that starts at a maximum of 9 and can only be reduced during the game. Let us say Sir Jon has nine magic wishes granted to him at birth. We also add A (5)=9 to the array, remembering that we also have to change the DIM and FOR . . . NEXT statement. Sir Jon will be able to use these wishes at crucial points in the game.

Sir Jon's aim in this adventure will be to become a knight of the Round Table. Not only can we use the variables which make up the character to calculate each situation as it occurs but we can also use them to decide when Sir Jon is worthy of becoming such a knight. We would probably make the test easy to start with and complicate it as the game is developed. So let us say that if Sir Jon has knowledge of 8 or more and his strength is 8 or more then he can become a member of the hallowed order of the Round Table. We can express this as a single line of BASIC:

```
IF A(1)>=8 AND A (2)>=8 THEN roundtable =1
```

The variable 'roundtable' is being used as a flag, i.e., a variable marking whether Sir Jon is a knight of the Round Table or not. If the flag is set to 1 then he is; if it is set to 0 then he is not.

The main aim of Igor will thus be to increase strength and knowledge, while his subsidiary aims will be to increase skill and constitution in so far as they help him in his main aim. At the same time he will want to preserve his nine wishes for the most vital moments. The rewards of the program will thus be increases in these variables.

However, we can also add a more abstract points reward for those players who like such things. We can invent an overall score dependent on how well the character is doing. In the case of Sir Jon it seems important that a knight should behave as honourably as possible, so we will give him honour points depending on how well he does in particular situations. If we make this scale 1 to 100 and record this as A(6) (remember to change those statements) we can also incorporate this rather abstract score into the test for knighthood. Suppose that to be admitted to the Round Table a knight must be very honourable, with an honours total of over 90, but that his honour is worth more if he has not used his wishes to aid him—he has done it under his own steam and not with supernatural aid. We will say that for every wish he has left he scores five honour points. This means that total honour points will be those normally added to A(6) plus 5* A(5). Our test for knighthood has now become a short routine:

```
7799 REM KNIGHTHOOD TEST
7800 HP = 0: REM CLEAR ANY PREVIOUS VALUE
7810 HP = (A(5)*5)+A(6)
7820 IF A(1) >= 8 AND A(2)>= 8 AND HP>90 THEN
ROUNDTABLE = 1
7830 RETURN
```

Line 7810 has more brackets than some BASICs might think necessary. Different versions of the language have different priorities for evaluating expressions so it is best to keep the expression explicit.

In the puzzle adventure game the character is less important than in the combat game. Usually the character has no variable characteristics. Instead the player's persona varies only according to the objects or items that have been collected. In effect the difference is that whereas in the combat game the persona has a constant set of attributes whose actual values vary, in the puzzle game it is the attributes themselves which vary, with each particular attribute having a constant value.

For example, the puzzle may involve finding an apple (in order to bribe a teacher), finding a bomb (to blow a hole in a door) and finding a coin (to get past a guard). Each of these objects, a, b, and c, has a constant value in terms of the program: a has the value 'enables bribe of teacher', b has the value 'opens door', and c has the value 'gets past guard', but the character may carry any combination of these at a time, e.g., a+b, a+c, b+c, etc., which means that the tasks the character can carry out successfully at a particular time will vary, just as in the combat game.

An array could be used to hold this information just as in the combat game. Each variable in the array will be used as a flag to represent a particular object. If the correct flag is set to 1 the character possesses that object, if to 0 he or she does not. However, as these flags can be set to more than two values they can be used for other purposes, to indicate various states of the object in question. After all, any particular variable in a program uses at least one byte of memory and not just one bit. A bit of information is effectively a flag which can only be set to 1 or 0, i.e., it has only two possible states. In other words it is binary, and bit stands for 'binary digit'.

However, a byte can have 256 different states, which is why many aspects of BASIC are limited to 256. If you look at the BBC/Electron character sets in the appendices of the manuals you will see that these consist of 256 codes (some of which are not used). A byte is made up of eight bits which allows coding of numbers up to 256 using the binary system of counting. Consequently, any variable which can be set to 1 in a BASIC program can also be set to 255 (the 256th state being 0, which is also a number). So if we use such flags in a puzzle game, we could use a system like the following:

0 means the object is hidden.
1 means the object can be seen.

2 means the player has the object.
3 means he has used it correctly.
4 means he has put it in the correct place.
And so on up to 255.

The combination of such a set of flags thus amounts to a description of the current level of achievement of the character, i.e., the stage reached in solving all the problems. So if your game is primarily a puzzle game the way you design your character is to consider all the puzzles that are to be solved and what set of flags will best record all their possible stages. It is certainly possible to have a different flag for every possible state or stage, but that would be very wasteful. Instead we might decide that there are eight main puzzles, each of four stages. Though it would be possible to record this information with only two flags (using methods we cannot discuss here) it is easier to have eight with four possible states (0 to 3) rather than 32 different flags.

6.3 Monsters

If the game is to have a combat element then there have to be opponents for the player's character. Even if there is no combat there will probably be beings of some kind which the character will be able to interact with, especially if we are building in intelligent processes of language. To make things easier we will regard all such creatures as monsters, even though some might be perfectly friendly humans. In my experience no one in an adventure has any real interest in helping the character—they are all in it for some monstrous purpose of their own. Monsters could be anything from rustlers to dragons to Klingons—it is a catch-all term for any creatures not controlled by the players.

A monster will be a configuration of numbers similar to the player's character. It may contain exactly the same range and type of variables as the character, but usually will have fewer, the range and type being a reflection of and related to a subset of the character's attributes. For example, if the character has an attack rating, a speed rating, and a treasure variable, the monster might have a defence rating, a speed rating, and a hoard of treasure. The combat routine will then depend on the relations between character attack and speed and monster defence and speed, with the reward for winning the combat being the monster's treasure, or more points, or both.

As monsters are one of the key hazards in this type of game, providing much of the interest, much thought should be devoted to their design, bearing in mind the following criteria:

1. A monster should only have variables and values which are meaningful in terms of the rest of the game. This usually means that those variables are related to the player character variables, but in the case of an 'intelligent' monster which can act in the program independently of the character, other variables will be needed.
2. The character should seldom encounter monsters which are extremely powerful, comparatively speaking, unless as the result of a major mistake (otherwise the game becomes 'sudden death').
3. No monster should be invincible, or too easy to defeat! Actual monsters encountered should either be in a range of powers, all of which can be overcome by the character, or should have their powers related to the current powers of the character.
4. Each monster should be different, not simply by virtue of the magnitude of its variables but also in terms of its overall configuration, i.e., its name, its behaviour, the kind of problem it presents, and consequently the choices/strategy needed for its defeat.

Broadly speaking this gives us two kinds of monster—the monster that is randomly encountered and the intelligent monster. The random monster can be found at any suitable location, but the intelligent monster will only be called by the program if certain conditions are met. Random monsters are relatively fixed in their function—it is to respond to the player, usually in combat. Intelligent monsters may be programmed with more complex behaviour and attributes which may lead them to have purposes independent of the character. They will have a 'personality' in some sense and the player character will have to interact with them.

Let us work on the random monster first, as this is the usual type in the majority of adventures and is the easiest to design. We can then consider the differences which need to be programmed into intelligent monsters.

We can hold the data, the numbers describing each monster, in any of a number of ways. The three easiest in BASIC are the array, the character string, and the DATA statement. We will look at each of these in turn before considering the exact data we are going to store. We will asume a set of data made up of the numbers 1, 2, 3, and 4. To hold such data in an array of one dimension would mean a different array for each monster, which is a possible though usually clumsy and wasteful method. It would be better, therefore, to use a two-dimensional array in which one dimension holds the list of monsters and the other the data for each of those monsters.

If our monsters were Bugblatter Beast, OOzler, and Giant Turnip and their respective data was 1, 2, 3 and 2, 3, 4, and 1, 3, 2, then our array can be thought of as the following matrix:

Bugblatter Beast	1	2	3
Oozler	2	3	4
Giant Turnip	1	3	2

The column here represents value and the row each particular monster. If we called the array mon, then a BASIC routine to create and fill such an array would be:

```
10 DIM mon(3,3)
20 FOR I = 1 TO 3
30 FOR J = 1 TO 3
40 READ mon(I,J)
50 NEXT J
60 NEXT I
70 REM BUGBLATTER BEAST
80 DATA 1,2,3
90 REM OOZLER
100 DATA 2,3,4
110 REM GIANT TURNIP
120 DATA 1,3,2
```

To find the appropriate piece of data at any stage in a game the program needs to know two things in addition to the name of the array it is to consult. These are the number of the monster being looked at (the row of the array) and the number of the value required (the column on the array). If we were calculating a combat and needed to know the attack value of the Oozler, we would use mon (2,3), the third item in the second row.

Arrays are very useful for this type of procedure, a process usually known as random access because any random item randomly selected can be accessed as easily as any other. However, arrays use memory and if there are a large number of monsters in the game an array to hold them all would use a great deal of memory, which would be particularly wasteful if the values were only in a small range, such as the 1 to 4 range above. The array mon will use at least 20 bytes on most systems. On the BBC and Electron 45 bytes would be needed to create the above array. This may not seem much but if you have 20 monsters you will need 20 × 3 × 5 bytes, i.e., 300 bytes. Some can be saved using integer arrays or byte arrays as used in

Dilemma, and we can actually cram nine numbers in the range 1 to 4 into only three bytes because four values can be held in two bits and 9*2 = 18 bits or a little over two bytes. In the adventure in Chapter 9 you will see how we can hold several pieces of data in one byte in this way.

One other way to save some memory is to hold the monster data as character strings. Such a string can be declared at the beginning of the program just as we might dimension the required array. For our current example the declaration of the string would be:

```
M$="123234132"
```

This string would only occupy nine bytes, much less than the equivalent array. However, to access the information in the string requires a more complex procedure than just referring to an array subscript.

In the first place we need to know in which section of the string the required monster data are held, then we need to know which piece of data in that section is required, and finally we have to turn the string data into numerical data which can be used by the program. If we are interested in the Oozler's attack value, we want section 2 (a substring of three characters), item 3. We have a look along the string in groups of three characters at a time till we reach the second group and then look within that group till we find the third item. BASIC allows us to create a string function which can do this job. The function could be defined as:

```
DEF FNk(M,I) = VAL(MID$(M$, (M*3)+I,1))
```

M is the monster number and I is the item number. When we want to find the Oozler's attack we write:

```
attack=FNk(2,3)
```

This might not appear too complex. However, writing to such a string is more difficult. If we wanted to alter the Oozler's attack value within the program and it was stored in array mon, we need only write:

```
mon(2,3)= x
```

where x is the new value. However, using a character string we have to find the correct item, delete it, and insert the new item, having turned x into a character. Again this could be done by a few lines of program used as a subroutine, but functions can also do the job:

```
DEF FNa$ (M$, M,I) = LEFT$ (M$,((M−1)*3) +I−1) + STR$(x)
RIGHT$ (M$,(LEN(M$)−((M−1)*3) +I +1))
```

It is not necessary to define all parameters for this function because some are already defined in the rest of the program—M$ is the string of characters we are using, M is the number of the monster and I the number of the item we want to change, and X is the new value.

This complex function takes the string to the left of the required item and adds that to the string version of the new value, adding to the result the remainder of the string beyond the old value. The old item is therefore cut out of the string by selecting halves before and beyond it, while the new value is inserted between them, in the equivalent place.

Although a complex procedure, it is probably worth while if you are handling large amounts of data, and memory is crucial. Storing data in this way does have two other defects, however. Firstly, in an array random access means that it takes virtually the same time to find any piece of DATA, whether it is held at the start, the end or the middle of the array. However, to process the information in a string, particularly if using nested loops, but even with functions that access information in sequence, the further the desired information is in the string, the longer the search will take. Consequently, it is a good idea to store the data which will be used most at the beginning of such a string and that used least at the end.

In order for the functions to work properly the string has to stay the same length and the data cannot move position. This means that the original string has to be the maximum length needed by the program, and all positions in it must be filled. If the values of particular items are likely to change from 1 to 20, each slot in the string has to be treated as a two-position slot from the beginning. This is true even if only one item in the string will be two characters long. So our example string would now be “010203020304010302” and all functions would have to operate on items two characters long rather than one.

One way to get around this problem, if we do not mind our data being somewhat limited, is to use characters instead of numbers in the string. Each character in a computer’s character set will have a distinguishing code. For the BBC/Electron these are returned by the function ASC. As most of these codes are two- or three-digit numbers, storing a string of single characters is the same as storing a series of numbers between 32 and 255 if we look at the codes of the characters rather than their display values.

Unfortunately the codes up to 32 are usually control codes and are difficult to manipulate in such strings, so if we want a range of numbers starting at 1 we have to subtract 32 across the board. We would also want to avoid the " character as this may cause problems with string handling, so we start our useful range at ASC 35, giving us an actual range of 1 to 221. This is nevertheless a useful improvement on the clumsy handling that number strings involve. Our sample string would now be "#$%$%&#%$". To decode it we use a routine like the ones above, but substituting ASC for VAL. For example, to represent 1 we must add 34 to 1 then turn it into the character equivalent of that number using CHR$—CHR$ (35) is #.

To write to the string we use function a$ above, again substituting ASC for VAL. In both cases we must remember to subtract or add 34 to turn the actual range into the allowed range.

Our final method of data storage involves direct handling of DATA statements. You may have noticed that in getting our data into an array earlier in this section we read the data from data statements. This means that, in a sense, the same data are held twice in the program—once in the array and once in the data. The reason that arrays are used is that they can be manipulated with ease, whereas string handling can be more complex and data statements cannot be manipulated at all within a normal BASIC program.

Consequently, data can be read from DATA statements but not written to them. If our program is such that we do not want to manipulate these data (or to manipulate it only temporarily and not store the results), we might find that an array is a waste of time and coding, and simply read from DATA instead. Some applications of this are discussed in other chapters. However, the method is simplicity itself.

Every monster is given its own DATA statement on a separate line of the program and the data are held in a known order, corresponding to the fixed order of our array or string. To find the particular piece of information we want simply RESTORE the DATA pointer to the correct line number and READ DATA in to a single variable until we have READ the correct number. If, for example, our Oozler's DATA are stored thus:

```
1010 DATA 2,3,4
```

the following short routine finds the attack value:

```
50 RESTORE 1010
60 FOR I = 1 TO 3
70 READ A
80 NEXT I
90 PRINT "OOZLER'S ATTACK VALUE IS"; A
```

As you will see in The Opal Lily in Chapter 9 one of the methods of storing numbers used to save memory for holding descriptions and messages is a combination of holding numbers as coded strings and keeping them in DATA statements to be read in this way. Although this program manipulates a large amount of data it uses no arrays at all. Instead, almost all the data have been placed in memory as a block of bytes by another series of programs, the memory block itself has been saved, and this large memory block is loaded and used by the main program.

Having looked at how we might store the monster's characteristics, let us briefly look at what those characteristics might be.

It is generally better to give the monsters a range of abilities as well as a range of values. In a fantasy game this means such devices as giving them magical powers, special forms of attack or behaviour, and different descriptions. In a more realistic game, such as a wild west adventure, personalities might be developed for different 'monsters' as well as giving them a range of skills (such as lassooing, shooting, wrestling, rustling, drinking, gambling, etc.). For intelligent monsters such differences are crucial.

The aim, of course, is to give variety so that the player does not have a good idea of what to expect and always finds something new about a particular game. Any feature which can alter player-monster interaction is worth considering for incorporation—perhaps different monsters communicate in different ways; perhaps after befriending a monster by offering gifts or talking, the player may be accompanied through the adventure; perhaps monsters of different ages, sexes, heights, weights, or religions may be sensitive to certain kinds of remarks; perhaps some monsters have other friends or enemies within the adventure; perhaps some monsters know information about others; and so on. Almost any feature that can be found in a real-life encounter or a novel can be programmed into a game by setting up an appropriate routine and a database.

You will find in the next chapter a short program which should stimulate your imagination in this direction. What is important to remember, though, in turning any of these ideas into code, is that there must be a balance achieved between the amount of program (memory, time, coding) that is required and the effect of the monster on the game. If half the program is used simply to generate one clever monster, found in room 100, the player is unlikely ever to appreciate the intricacies of your design.

6.4 Objects

In an adventure program objects are of two kinds—portable and fixed. Portable objects can be moved from location to location in the game, but fixed objects cannot. In game terms this means that the fixed object is essentially a feature of a particular location, an aspect of its description. From the player's point of view there is no difference between the output “You are in a dark and smelly tunnel” and “There is a brass candlestick hanging on the wall” if no input can have any effect on the description. From now on, therefore, we will use the word ‘object’ to refer only to items which can be moved from location to location and we will regard fixed objects as features of particular locations.

Objects serve several purposes in adventure games. In terms of player psychology they provide the immediate rewards (you will remember that we previously analysed the game as being a strategy designed to gain rewards). In other words, each time the player first finds an object which can be taken or manipulated in some way a little victory has been scored or part of the game has been ‘won’.

In terms of the narrative realism of the story which forms such a game, carrying things around is a key aspect of a plot, though sometimes the thing carried is information or ability rather than an object.

In terms of the program structure, having objects is one of the simplest ways of adding complexity and variety to our game. After all, a small game of 10 locations and 10 objects could give 100 possible events in the game; a large game of 1000 locations and 100 objects can give 100000 events, which is probably more than even the most dedicated Igor would care to solve.

An object may have no significance in the game, being some kind of red herring, time-waster, or obstacle, but usually it has a more specific purpose. In the puzzle game it will usually form part of the solution to the puzzle, e.g., to find the missing formula Igor must enter a room with a locked door, so first he must find and bring the key. In the combat game objects normally improve the character's abilities, giving more efficiency in some other aspect of the game. For example, finding a shield may improve the defence value and hence the likelihood of surviving future combats; acquiring a magic potion may increase the character's strength, and hence attack value, and/or ease of carrying other objects.

However, certain penalties may also go along with such advantages. In puzzles it is often the case that having found an object needed to get past one problem, finding the object itself creates another problem. Finding gold dust may help you bribe the bartender, but how do you now get past the bank robber? Finding a two-handed axe adds eight to your attack value, but you cannot now carry a shield so your defence goes down.

From the programming point of view, therefore, an object can be regarded either as a flag, signalling that a particular condition is met (the object is found) and therefore certain consequences are permitted, or as a function which acts on one or more of the character's variables, usually by increasing them. This means that objects may be represented in our program in several ways, and one object may have several representations. It also means that, just as the construction of monsters depends on the overall program structure, and particularly the character's structure, so objects must be thought of as extensions of the character, modifying it according to built-in rules which allow certain development in the game to take place.

Let us explore an example. Suppose we wanted a game in which the player was a nineteenth century explorer in Egypt. One of the key problems we can give Igor is the deciphering of hieroglyphics (a good opportunity for some interesting graphics). We would want him to start off with minimal skill in decipherment, which he would then be able to build up. He starts with D=10, which gives him a 10 per cent chance of understanding any hieroglyphic he encounters. If, however, he finds the Rosetta Stone he will have a dictionary of the majority of the signs, so his chance goes up to 60 per cent.

Such ideas can easily be complicated without much extra coding. Inside a pyramid or tomb, because of the darkness, the chance goes down to a quarter of its normal rate, unless a lamp is used, in which case it goes down a half of the rate. If the player is able to decipher a magical hieroglyphic then his chance goes up to 100 per cent for a limited period, but if he commits sacrilege it drops 5 per cent. Unfortunately, the Stone is very heavy, so cannot be carried at the same time as any treasure, and if a mummy sees the Stone it will immediately attack the bearer.

In this way a complex story potential is built around a few simple variables, and each element is fitted together with all the others. The monster and objects are related (mummy and stone); character abilities and objects are interrelated; situation may affect all of these; and success can lead to further success. Yet there are possible penalties and pitfalls and the reward itself might be a peril (such as in successfully using the Stone to read a curse which is then activated).

As with monsters, it is therefore important that objects should not only make sense in programming terms but they should also make sense in plot and game terms. Good adventures are not a series of arbitrary actions. Nor should they be a series of highly structured actions which appear to be arbitrary. Good games, from the player's point of view, are not those which are well written but those which appear well written. The effect of a game and its appearance is often more important than its actual nature or content.

7 TEXT IN ADVENTURE

As most adventures are mainly ways of processing text it is worth pausing for a moment to consider the textual aspect of adventures. One reason why many people like adventures is that they are a little like people because they 'understand' and 'use' real language. This is one of the distinguishing features of intelligence, of course, so we need to consider how a program can understand language. The adventure given in Chapter 9, The Opal Lily, uses only some of the techniques discussed here but they are all useful in adventure design for one purpose of another. This chapter presents the essentials but The Opal Lily adds some more sophisticated versions.

7.1 Input

The kind of input that an adventure game uses depends to some extent on its type. A real-time graphics adventure, for example, needs single-key entry, whereas a complex puzzle adventure needs input that is as close as possible to ordinary English. There are other types of input, but we will look at these two. The Opal Lily is primarily of the second kind, though some of its input can be simplified. In both cases the procedure is the same:

1. INDICATE THAT INPUT IS NEEDED
2. THE PLAYER INPUTS INFORMATION
3. CHECK THAT THE INFORMATION IS ACCEPTABLE
4. INTERPRET INPUT
5. ACT ON INPUT

Single-key input is the easiest to program and in some ways the easiest for the player if there are only a few possible inputs. However, if too many keys are required (more than about eight) the player will forget which is which and make mistakes. While this can be useful in some games, most players will dislike a game which is essentially designed to test their knowledge of the keyboard.

For single-key entry, numbers are easiest to process, particularly because numeric input is easy to check. For example, if the numbers

2 to 6 are the only allowed input, a routine such as this will do the job:

```
80 REPEAT
90 PRINT "Please type a number between 2 and 6 and press
RETURN"
100 INPUT A
110 A = INT A
120 A = ABS A
130 UNTIL A<6 AND A>2
```

Line 110 is necessary to ensure that players do not try clever input involving decimals. Line 120 similarly ensures that negative numbers are not typed in.

As each key on the keyboard has a code, we can define a range of allowable keys in the same way. The codes for all the BBC/Electron characters are contained in the back of the manuals, being standard ASCII and Teletext codes. For example, lower case letters a to f are ASCII codes 97 to 102 inclusive. To check this input we would change the routine above to read:

```
80 REPEAT
90 PRINT "Please type a letter from a to f and press RETURN"
100 INPUT A$
110 A$=LEFT$(A$,1): REM JUST IN CASE MORE THAN ONE
LETTER IS TYPED
120 UNTIL ASC(A$)>97 AND AS(A$)<102
```

The problem with this is that the keys we choose are not likely to be easy to remember unless each letter is the first letter of the command it stands for, such as the usual N,S,E,W for the points of the compass. Checking for errors in input as complex as these can itself be quite complicated if we rely on the number codes, because error-checking of this kind relies on allowing only numbers in a set range. Therefore we add a different kind of checking to our routine, ensuring that only input of an allowed letter is possible.

No matter how clever our program, there will be some possible inputs we have not allowed for and the player can always make a mistake. Our error-checking has therefore not only to make sure that the input is in the correct range but also that it is sensitive to other possible errors. Every time the player presses a key there should be an appropriate routine to trap any errors. The error trap should come early in the routine if a mistake is likely, but may come at the end if mistakes are less likely. If in doubt the check should come immediately after the INPUT (or GET($) or INKEY($)) statement.

If an error is detected then some form of error message should be sent to inform the player of the mistake. The best error messages tell the player three things:

1. That an error has been made
2. The kind of error that has been made
3. The kind of input that is acceptable

The worst kind of error message is no message at all! The player will have no idea what has happened or what is meant to be done.

If we are allowing a number of different single-key inputs all we can do to check them is to test for each one in turn. We could do this mechanically by using an IF . . . THEN statement for each of the possible inputs each time input is required, with the default line being 'ERROR', but this has the usual faults of inelegance, wastefulness, and inefficiency.

What we want is a routine to check that the input letter is one of an acceptable set of characters. It is quite likely that at different points in the game different sets of characters will be allowed. For example, during combat the possibilities might be:

A = ATTACK
M = MAGIC
R = RETREAT

whereas on entering a new location the options might be:

L = LISTEN
R = REST
S = SEARCH
W = WAIT

Therefore the routine should be a general one which can be called whenever single-letter input is expected to test for its validity.

The obvious method for us to use is to hold the valid characters as a string and compare the input character with each of the characters in the string in turn. We will call the string test$. Then every time single-letter input is expected we use GET$ rather than INPUT, to maximize the main advantage of single-letter input—which is speed—using the following routine:

```
  80 REM FOR COMBAT INPUT
  90 PRINT "Please type a single letter (A,M or R)"
 100 test$ = "AMR":REM DECLARE TEST STRING
 110 PROCcheck
 120 PRINT "O.K. That´s valid"
 900 END
 999 REM CHECKS FOR VALID CHARACTERS
1000 DEF PROCcheck
1010 REPEAT
1020 A$ = GET$
1030 IF INSTR(A$,test$)=0 THEN
     PRINT "No, you idiot, ";test$
1040 UNTIL INSTR(A$,test$)<>0
1050 ENDPROC
```

If you follow this routine through you will see that control only returns from the subroutine if one of the correct keys is pressed.

PROCcheck can be called at any time in the game so long as we remember to declare the test string before the call is made. It is important also that we do not include any invalid characters in the string, such as punctuation or spaces, unless these are possible valid input.

To make such a routine efficient we should also ensure that any lower case input is converted to upper case (or vice versa), thus making the test strings much shorter and easier to code. To make it friendly it should tell the player the character which has just been typed so that any mistakes can be monitored. One of the worst features of games players is the way they blame the program for the way their fingers slip around the keyboard.

If we use single-key input we should try to do the following:

1. Choose only a few keys.
2. Choose keys whose meaning easily stands for the appropriate command.
3. Make sure we can check for potential errors in the input.
4. Choose a combination of keys that can easily be handled on the keyboard, especially if the game is real-time.

What about 'English' input? It is usual in text-based adventures for commands of word pairs to be allowed, such as "GO WEST", "TAKE BOOK", etc. These are obviously more complex than single-key ones—they take more time to design, to code, and to use in play, but they offer much more varied rewards than limited single-key input.

Even with this complexity a technique similar to that for testing single-key input can be used. The differences are that INPUT has to be used instead of GET$ (so the RETURN key has to be pressed after each input by the player) and the string which is input has to be divided into its separate words. We will assume a two-word input, though the principles remain the same for longer phrases or sentences. Most adventures only allow two-word input, but in Chapter 8 we will discuss some ways this might be changed. We need to declare some string variables as well so we will use the following:

A$ is the string input by the player.
B$ is the first word.
C$ is the second word.
D$ is the first three letters of the first word
E$ is the first three letters of the second word.
noun$ is a test string for nouns.
verb$ is a test string for verbs.

It will become clear why all these are needed. The algorithm we will use is:

1. PLAYER INPUTS TWO-WORD COMMAND
2. WORDS ARE SEPARATED
3. D$ IS SET TO THE FIRST THREE LETTERS OF THE FIRST WORD
4. D$ IS CHECKED AGAINST V$
5. E$ IS SET TO THE FIRST THREE LETTERS OF THE SECOND WORD
6. E$ IS CHECKED AGAINST N$

Although we could check the complete words that were input this would take more time and memory than is necessary. A three-letter code is enough to distinguish many words and, provided we can ensure that no pair of valid command words in our program begins with the same three letters, there will be no problems—do not pick CANDLE and CANNON in the same program. If, for some reason, you need to have keywords which start with the same three letters, you will have to use four-letter codes (or more, if more are needed to distinguish the words), but this should be avoided if possible.

Suppose we have an adventure which uses two-word input. Among its allowed commands are: LIGHT TORCH, TAKE WHEEL, and SWIM RIVER. The allowed verbs are thus LIGHT, TAKE, and SWIM, while allowed nouns are TORCH, WHEEL, and RIVER. Notice that no two of these words begin with the same three letters. We can compile strings made up of the three-letter codes of acceptable words in the same way that we did for test$ in the single-key input routine above. Thus somewhere in the program will be a line saying LET N$ = "TORWHERIV" and another saying LET V$ = "LIGTAKSWI".

We can then compare the first three letters of our input words (D$ and E$) with each three-letter section of this string and, if it matches, we can then branch to the appropriate routine. Let us work through the whole procedure a stage at a time in BBC BASIC. Call the procedure PROCdecode.

First, input the words:

```
999 REM TWO WORD INPUT
1000 PRINT "What next?"
1010 INPUT A$
```

Then separate A$ into two words:

```
1020 REPEAT
1030 IF INSTR(A$," ")=0 THEN PRINT "Two words please"
1040 UNTIL INSTR(A$," ")<>0
1050 L = 0: B$="": C$="": D$="":  E$=""
1060 FOR i = 1 TO LEN(A$)
1070 IF MID$(A$,i,1)=" " THEN B$ = LEFT$(A$,i-1): C$ =
     RIGHT$(A$,(LEN(A$)-i))
1080 NEXT i
1060 ENDPROC
```

Next we take the first three letters of each of the two words. The problem here is with words such as GO which have only two letters. We have to add a space to these, which means that V$ must include GO plus space if it is an allowed command, e.g., "LIGTAKGO SWI". Line 1060 does this:

```
1060 B$ = B$ + " ": C$ = C$ + " "
1070 D$ = LEFT$(B$,3): E$ = LEFT$(C$,3)
1075 FL = 0
```

In a similar way we can use a string such as "N S E W " to define movement commands which can be input as single-letter commands but will not be confused with the initial letters of other words.

Now we check D$ against the verb codes (V$) and E$ against the noun codes (N$). As the process is the same for both we can use the same checking routine twice, provided we declare the relevant strings before we call it.

```
1080 LET T$ = V$: LET U$ = D$: PROCword
1085 IF FL = 1 THEN ENDPROC
1090 LET T$ = N$: LET U$ = E$:PROCword
1095 IF FL=1 THEN ENDPROC
```

```
1096 PRINT "O.K"
1099 ENDPROC
1199 REM PROCEDURE TO COMPARE TEST STRING     AND
     AND INPUT WORD
1200 DEF PROCword
1210 FOR I = 1 TO LEN(T$) STEP 3
1220 IF MID$(T$,I,3) = U$ THEN FL=I
1225 NEXT I
1230 IF FL<>0 THEN ENDPROC
1240 PRINT "I don´t know how to ";A$:  FL = 1
1250 ENDPROC
```

Note the flag FL set at lines 1075 and 1240. Line 1075 sets it to 0, in case it already has been used. Line 1240 will set it to 1 if the first half of the command is invalid so that, on returning to 1085, the procedure is not run for the second part. Notice also the STEP command in line 1210, which may be an instruction you have not used before. This simply means that the FOR . . . NEXT loop between lines 1210 and 1230 will increase I by three at a time, thus moving along T$ three characters at a time, which is how our codes are grouped. Normally a FOR . . . NEXT loop will operate in steps of one, which is the default value if STEP is not specified.

There is a slightly more efficient way of doing the check in lines 1210 and 1230 by using the INSTR command (which is not available in a number of BASICs). We can replace lines 1210 to 1230 by the single line:

```
1210 IF INSTR(T$,U$) THEN ENDPROC
```

Having checked that the input is alright we now have to branch to the appropriate routine in the program. The most elegant way to do this would be to use numerical input or character codes as the basis of sending control to subroutines. For example, a very elegant input and selection routine would be:

```
90 PRINT "Please type an instruction from 1 to 9"
95 REPEAT
100 INPUT A:A = INT(A)
105 UNTIL A>1 AND A<10
110 GOSUB A* 1000
```

This sends control to subroutines beginning at lines 1000, 2000, etc., up to 9000. However, such elegance is not always possible if the input is initial letters or full words. Nor is it possible to use conditional control like this for PROCedures. If we wanted to send control in this way to a procedure rather than a subroutine in BBC BASIC we would have to use a series of lines like:

```
110 IF A = 1 THEN PROCa
120 IF A = 2 THEN PROCb
130 IF A = 3 THEN PROCc
```

By careful design we could send control to a subroutine related to the ASCII character code or the sum of the codes in any three-letter combination, but the design effort is probably not worth while. We may have to resort to a series of one-line tests, whether procedures or subroutines are used, of the form:

```
IF input = keyword x THEN GOSUB routine y
```

So for our single-key example appropriate lines might be:

```
130 IF A$ = "A" THEN GOSUB 2000
140 IF A$ = "M" THEN GOSUB 2100
150 IF A$ = "R" THEN GOSUB 2200
```

and for two-word input appropriate lines might be:

```
130 IF B$ = "TAK" THEN GOSUB 2000
140 IF B$ = "SWI" THEN GOSUB 2100
```
etc.

However, BBC BASIC can go one better than this to make things somewhat easier. The parameter of a GOSUB or a GOTO can be a numeric variable and numeric variables may be any number of characters. Consequently each acceptable letter or three-letter code can be used as a variable name and the value of the variable can be declared as the line number of the appropriate subroutine.

To clarify this let us use the example of SWIM and suppose that the swimming routine begins at line 2110. During initialization of the program we can assign a value to the variable called SWI, which also happens to be our three-letter code for 'swim'. When input has been through all the validation checks B$ will hold "SWI" as the first part of the first word. If we can turn the string held by B$ into the value held by the variable with the same name, we can use the input string to direct control.

The function which does this is VAL, which turns a string into its appropriate value. If our program contains among others the following lines:

```
10 LET SWI = 2110
..........................................
140 GOSUB VAL (B$)
..........................................
..........................................
2110 REM SWIM ROUTINE
..........................................
..........................................
```

it will work if "SWIM" is the first word input.

The advantage of this is that it is easy to follow the workings of the program and it saves coding. We do need to declare a variable for every acceptable input word, but we can dispense with all the long-winded IF . . . THEN statements. It also allows clever programmers to alter the routine a particular command is running. For example, there may be two locations in which a player might be allowed to swim, namely, a placid stream and a dangerous river. If the player swims in the stream there is no danger of drowning. In the river, everything is the same as for the stream, except for the danger of drowning. So the same routine will be used for each, with an extra piece of code for the river. If this extra piece of code was held at line 2005, with the common routine at 2110, then a line like the following could make use of it:

```
135 IF LOCATION = RIVER THEN LET SWI = 2005
```

However this trick will only be of use in a few circumstances, as extra code can be held and called in other ways, e.g., by the use of flags or nested subroutines.

As the most frequently used nouns will be those for movement we can speed things up by having two noun strings, one full of objects, the other for directions, e.g., "NORSOUEASWES", and only test the latter when a valid movement verb is detected, e.g., in the case of most adventures the verb GO. In addition we may need two special commands which do not need nouns, namely HELP and INVENTORY. The first of these gives help to the player with problems, the second lists all the objects the player currently has. So before the standard word comparison routine we would need three separate tests for these three verbs. As the latter two have no nouns, we test for these before the main routine, i.e., between lines 1015 and 1020:

```
1016 IF LEFTS(A$,3) = "INV" THEN GOSUB INV: RETURN
1017 IF LEFTS(A$,3) = "HEL" THEN GOSUB HEL: RETURN
```

The test for GO will come before the main verb and noun tests, i.e., before 1080. If W$ is the string "NORSOUEASWES" then lines 1078 and 1079 will do the job:

```
1078 IF D$ = "GO" THEN LET T$ = W$: LET U$ = E$: GOSUB
1200: IF FL = 1 THEN RETURN
1079 IF D$ = "GO" THEN GOSUB VAL (B$): RETURN
```

These check that the word following GO is a legitimate direction, send control back to the main program if not, or forward to the GO routine if it is.

We must also minimize the effects of faulty input of commands. Likely mistakes involve typing only one word, or words of less than three letters, or extra spaces. An easy way to avoid most of these problems is to declare the variables not as empty strings but as strings of spaces and to clear each string to the required number of spaces each time the input loop is called.

Once the loop has run it will return with a flag, marking an error if one has occurred, and input will again be repeated. However, if both input words are correct control will have to be passed to other routines in the program. In order that the detected verbs and nouns can be used in such a control they have to be given numerical values. If we do not want to use the method already outlined of declaring variables with the same names as the verb codes and values equal to the subroutine line numbers we have one further alternative.

For every verb or noun we already have a value, namely the position of the three-letter code for each word in the relevant string. This is counted by the variable I in the loop which attempts to match D$ or E$ against these codes; however, I is used as the variable in several other loops, so monitoring it might become confusing. Consequently we can add a line which sets another variable, say K, to the same value as I if a match is found in the checking loop. Then K will be passed back to the main checking routine where it is turned into the appropriate variable, which we can call VERB or NOUN. If we use INSTR rather than a loop to check that an input word is allowed then K will be set to the position of the substring within the test string, i.e., the word number.

If there are special words to be detected then VERB and NOUN may have to be set specially, such as in the case of INVENTORY and HELP. If these are looked for as special cases before the general checker, VERB will have to be set specially.

With these two variables, VERB and NOUN, most of the descriptive routines can be called. VERB can be used as the control variable to call the appropriate verb routines and NOUN as one of the variables which determines the exact subroutine which is used.

As these are crucial values it is very important to approach them from a logical point of view. While coding my adventures I use at least three major reference sheets to find my way through the program. These are doubly important if you have no printer so cannot obtain hard copy of the problematic or unfinished areas of your program.

My first reference sheet is a map of the logical maze with each location numbered. Each location which is either a GET or NEED square has G or N written in it, plus the noun or nouns which are there. In addition any special puzzles or features are also noted here, such as a river which has to be swum. Finally, an H is placed in each square where an object is hidden, rather than being immediately obvious.

To make things easy I use the variable ROOM (or LOCATION) in the program to hold the current code for each location on the map, and this is used as the control variable for the description routines in the main block of the program. The correspondence is as follows: the location number is on my hand-drawn map, is held above HIMEM in a specific address (or wherever the block of data for the map is kept), and when multiplied by 10 and added to a base number gives us the number of the description routine to be called. The description routine would either be a subroutine or a DATA statement to be read in and PRINTed using RESTORE.

The second list is of all the verb routines and their start lines. These should be regularly spaced (at least 20 lines should give room for alterations) and kept in the same order as the verbs in the verb string. Consequently if you decide that a verb is not to be used after all or a new one is to be added you can see at a glance where it should go in the verb string and in the program, without having to worry about keeping track of how many there are or what VERB should be set to. (As usual put the most frequently used verbs first in the list, early in the verb string, and early in the program block.)

Finally, there is the list of nouns. This is probably the most important of the lists. It holds the names of all the objects and the solutions they are part of, as well as the line numbers of the verb routines which use them, and should correspond to the reference number of the noun in the noun list. The nouns are listed in the order they are held in arrays or memory, which roughly corresponds to the order in which the Gets and Needs should be encountered in the program in order to solve it. It is important to have a complete list because the noun routines, being the end points of a series of calls, can seldom be held in a very logical way and will therefore be difficult to locate during debugging. REM statements would help with this problem, but so many would be needed that the size of the program may well be doubled so they would have to be removed from the finished version. However, it is a sensible idea to create two versions of your program, one full of REMs so it is easy to understand and debug and one stripped to the bone which runs quickly and efficiently.

7.2 Talking to monsters

It is all very well killing monsters and stealing all their worldly goods, which is the standard fare in adventurers, but not every monster is hostile and some may be too powerful for lowly adventures to pit themselves against. Fighting in many circumstances would not come under the heading of intelligent behaviour. In such circumstances the wily adventurer is best advised to use sly flattery, gentle persuasion, high-sounding oaths, blood-chilling threats, or any other form of conversation which seems likely to win the monster round. However this means the monster should be able to talk back, and talking is a difficult thing to simulate.

A great deal of artificial intelligence research has gone into trying to make programs understand and produce recognizable conversation. Kenneth Colby's PARRY produces recognizably paranoid text. DOCTOR and ELIZA provide a number of conversational gambits like a psychiatrist. A program called SHRDLU is able to understand English commands for manipulating a world of coloured blocks and pyramids by using a robot arm. Some programs exist that can tell stories which are almost passable. However, by and large there are no programs which understand enough about the world to produce acceptable conversation.

This has happened for two reasons. Firstly, research is not sufficiently advanced for suitable programming techniques to be available and in some ways the available hardware is still limited. More importantly, linguists do not really know enough about the way human beings converse to be able to provide a suitable model for a computer. Whatever we do, it will be limited, but at the same time we can have the satisfaction that we might be breaking new ground and the problems that control our humble attempts at adventures are the same ones that some of the best brains in the world are currently puzzled by.

Two approaches are possible on a micro. We can aim for as full a simulation as possible in which many tests and transformations are made on the language, attempts are made at meaningfulness, and a wide and varied vocabulary is used. The program in Chapter 9 goes a little way towards this, although its cleverness lies in input rather than output. Or we can settle for a couple of sleight-of-hand tricks, which appear to allow conversation but actually just do random things with words. This has been the most successful approach so far, even in much more serious artificial intelligence research.

The first approach is very complex—too difficult to give a full account here. Useful routines can be adapted from accounts of recent research but in many cases implementation is too demanding for a micro, especially in the context of a program which needs memory for other tasks. The sleight-of-hand approach is usually based on two procedures—in the processing or input the program checks for keywords, by comparing each input word against a dictionary of recognized words and, according to the match that is found, will compile an output string with an approach meaning; and in the processing of that output some changes are made to the structure to make it fit grammatically with the input string.

An example of the first kind would be a test for a word like 'fight'. If this was found the monster might respond with "So you want to fight, do you?" and control might switch to the combat routine. An example of the second would be the transformation of pronouns, where if the first string is something like "I will give you my sword" then the output might be "O.K. I'll take your sword". Here 'my' becomes 'your' and 'give' becomes 'take'.

You will see that this requires a great deal of thought. The least that is required of a good routine to do this is that it should have a large and appropriate vocabulary, that it should produce reasonably grammatical sentences, that it should allow input of strings longer than two words, and that the response should be connected to the input. This means we would have to think of all the words likely to occur and find some appropriate response for each, which might demand much more work than the program justifies. However the essential design is easy to describe:

1. ACCEPT INPUT STRING

2. SEPARATE STRING INTO COMPONENT WORDS

3. HOLD ALL WORDS IN MEMORY

4. COMPARE ALL INPUT WORDS AGAINST THE DICTIONARY OF KEYWORDS

5. IF A KEYWORD IS FOUND THEN EXECUTE THE APPROPRIATE SUBROUTINE

6. IF THE SUBROUTINE USES THE ORIGINAL STRING THEN DO THE NECESSARY TRANSFORMATIONS ON THAT STRING

7. IF NO KEYWORDS ARE FOUND THEN EXECUTE A DEFAULT OUTPUT.

8. IF THE KEYWORD SENDS CONTROL TO ANOTHER ROUTINE THEN EXECUTE THAT ROUTINE, OTHERWISE GO TO THE BEGINNING OF THE CONVERSATION LOOP.

Stages 2 and 3 are simply expansions of the routines we have already used to analyse two-word input, namely looking for spaces and holding all items between spaces as separate words. Stage 4 will take each word in turn and each word in the database vocabulary and compare the two of them. This could be a very long process, especially if the input string or the vocabulary is large. It can be speeded somewhat by using a three- or four-letter code rather than the full word for matching (but this may result in faulty matches), by ignoring all words of less than four or more than six letters (because these are likely to be unusual or else serve grammatical but not semantic purposes) and by using an indexing system, based either on the alphabet or the codes of the characters, so that the search can branch through the database rather than look at every item.

For example, suppose the input string was "Give me your jewels or I'll chop off your head, you sycophant". In consulting our dictionary the simple method would be to take each word in turn and using FOR-NEXT loops compare it with each word in the dictionary. It is unlikely that any interesting responses can be built into the program to deal with 'or', 'off', 'me', 'you', or 'sycophant', which is why these are all outside our word length limit. 'I'll' could also be excluded because it contains internal punctuation and this could be tested for. This means a 12-word string is reduced to six. For each of these six the program then could do the word by word comparison, but if we arrange our dictionary in such a way that words can be compared alphabetically, such as by using a number of string arrays, the number of tests can be greatly reduced. If we suppose that only 'give' is held in this dictionary and the comparison is made alphabetically it would take only two tests to find a match, as the program first looks at 'chop' and then 'give'. In essence this is the approach taken in the use of a dictionary in The Opal Lily in Chapter 9.

Stage 5 is the heart of the program. For each keyword one or more possible outputs should be allowed. These could be randomly selected once control has been directed to them or they could be motivated in some other way. For our sample string there might also be some transformations used. For example, the 'give' routine could take all the words between 'give' and the end of the string or a conjunction (in this case 'or'), and invert any pronouns found in that substring; we get "give you my jewels". To this the routine adds "If I" and "what will you give me?". So the output formula is:

"If I " + transformed input string + "what will you give me?"

giving the perfectly meaningful response, "If I give you my jewels, what will you give me?" Note that to do this, the program only needs to recognize one keyword. It does not need to know what jewels are, nor who 'you' refers to.

The default output, used if no keywords are found, would contain a number of choices of non-commital remarks, such as "Tell me more", "That's easy for you to say", "I don't understand", and so on. Even with a first class string-processing program these remarks will be used more than any other, so a wide choice is needed to prevent too much repetition.

In principle, routines like these could be used throughout entire adventures and not just in exchanges with monsters, but in practice this has not been done, because many other potentials of adventures have not been realized—it takes too much time to code, too much memory is used, there are many problems with design of suitable algorithms, and no-one has seriously tried it.

However, the easier option has been used quite successfully. This involves more trickery than actual conversation, though the principles are quite similar to those just discussed. Here there is no attempt made to 'understand' the input string, but the monster makes a more or less random choice between a set of responses which make sense in the context.

For example, the player may be given a simple choice between fighting and talking to a monster. He chooses the latter because strength is low and says, "I'm very fond of goblins". There are many possible responses to such a remark—the monster could ask for more information, could become angry, could become very friendly, could be wary or deceitful. All of these possible attitudes can be expressed by phrases which are unconnected to the phrase which sets them off:

Tell me more.
So all you can say is "I'm very fond of goblins", is it?
I'm glad you've said that.
So that's what you think, is it?
You'll have to give me some time to think of an answer.

Being unconnected to the input string these can be chosen randomly and, providing the next choice is reasonably consistent, this will appear to make sense and could eventually lead to combat or to giving treasure, or to the monster stealing from the character,or any other action of the monster which is coded in the program. A sample routine for this which could be added to many adventures is below:

```
3800 R = 0
3805 PRINT "What do you have to say?"
3810 R = RND(3) + R
3820 INPUT A$
3822 IF R < 1 THEN GOTO 3950
3825 GOTO 3830 + (R*5)
3830 PRINT "I´m fed up with this, I´m going":
     RETURN :REM GOES BACK TO MAIN ROUTINE
3835 PRINT "So that´s how you feel is it?":
     R = R + 1: GOTO 3810
3840 PRINT "I don´t quite understand you":
     R = R - 2: GOTO 3810
3845 PRINT "Haven´t you anything better to say
     than that?": R = R + 1: GOTO 3810
3850 PRINT " So you don´t want me to eat you?":
     R = R - 3: GOTO 3810
3855 PRINT "Do you want to be friendly then?":
     R = R - 5: GOTO 3810
3860 PRINT "I´m getting impatient":
     R = R + 3: GOTO 3810
3865 PRINT "I detest adventurers like you":
     R = R + 2:GOTO 3810
3870 PRINT "I could give you a reward if you
     behaved nicely ": R = R - 5: GOTO 3810
3875 PRINT "I hate people who say things like that":
     R = R + 3: GOTO 3810
3880 PRINT "You´d best watch out for my temper":
     R = R - 1: GOTO 3810
3885 PRINT "You´ve one more chance to be pleasant":
     R = R - 2: GOTO 3810
3890 PRINT "I WARNED YOU!!!":
     GOSUB COMBAT ROUTINE: RETURN
```

```
3895 PRINT "Because I think you´re so rude
     I´m taking your treasure":
     (treasure variable) =0: RETURN
3900 PRINT "Stuff this for a lark ":
     (treasure variable) = 0:
     GOSUB COMBAT ROUTINE: RETURN
3905 PRINT "The monster works himself up into
     an apoplectic fit and expires on the floor":
     (treasure variable) = (treasure variable)
     + (monster´s treasure): RETURN
3950 PRINT "You seem so friendly I´m going
     to let you have all my hard earned savings":
     (treasure variable ) = ( treasure variable )
     + (monster´s treasure):RETURN
```

This works quite simply by increasing and decreasing R according to the output string chosen. As R goes down the monster is more likely to give his treasure away; as it goes up he is more likely to fly into a rage. However, if he gets really angry he may have a fit as a result, and if R ever becomes 1 he may just get fed up and go away. The strings are muddled up a little in order to make sure that R can fluctuate because monsters are notoriously temperamental. However, the random variable added each time will always tend to push him towards anger, so the longer the character talks, the more likely he is to annoy the monster.

(This is one example of 'poor spaghetti' programming which, because it is a self-contained routine and because there is always an exit sooner or later, is perhaps allowable. To write a similar routine in a structured way would actually be quite complicated. However, if you are going to write tangled code of this kind make sure it is well planned in advance.)

The routine can easily be expanded or contracted by adding and subtracting lines, and other outcomes can be added by using other values of R to direct control to other routines or by building in other variables. It would also be possible to incorporate simple checks on the player's input to affect these variables. For example, a dictionary of swearwords could be used as a database to check input, such that two of these words would immediately send the monster into a rage.

8 MORE ON ADVENTURES

8.1 Adventure maps

The key part of an adventure game is its map and the mechanism for moving the player's character around that map. We have already looked at how a graphic piece can be moved around a screen in Chapter 3. Now we will examine some of the ways of creating and representing a map and of moving a character around it. For most of the approaches discussed here it does not matter if there is meant to be a screen display or not, but some of the approaches are easier to implement with graphics than others.

It is common in adventure games as well as graphic games to control player movement by use of a set of keys. Normally these would be the cursor keys, or the arrow keys, with the arrows representing the chosen direction (these are the same thing on the BBC and Electron), or the keys N,S,E,W (for North, South, East, and West), or in some cases the number keys, especially if a numeric keypad is available. In practice any set of keys could be used and it is possible to define a 'mapping keypad', that is to say, a subset of the keyboard made up of nine keys in a square, with the orientation of the external eight keys representing the points of the compass and the central key either ignored or used for a special action, such as the ubiquitous 'hyperspace'. On a QWERTY keyboard the most likely mapping pad is the sector shown in Fig. 8.1.

$$\begin{bmatrix} Q & W & E \\ A & S & D \\ Z & X & C \end{bmatrix}$$

Figure 8.1

Having decided on the set of keys which will control player movement we must design a routine to interpret the keyboard.

8.2 Moving a character

The method used to move a player through the different events in a game depends on the type of game and the nature of the events. If our game has a constant graphic display we would probably use method C below. If, on the other hand, our adventure game is primarily textual we will probably use a method like that used in Chapter 9. However, the simplest methods are those I have called A and B: A is random movement and B is seeded random movement.

8.3 Method A: random movement

Whether our adventure is textual or graphic, we can make the link between successive events purely random. In this case there will be no map as such because returning to the same 'location' may well result in a different event. A typical structure might be as in Fig. 8.2.

```
1. PLAYER MOVES PIECE


2. GENERATE A RANDOM NUMBER BETWEEN 1

   AND 3 = R


3. CHOOSE A SUBROUTINE ACCORDING TO R


4. DO CHOSEN SUBROUTINE
```

Figure 8.2

Here only three types of event are possible, called routines 1,2, and 3. The choice of a particular routine is made only when the player makes a move but is random,i.e., unrelated to the actual move made. If the player moved south then north, i.e., returned to the same position, there would be only a third of a chance that the event that occurred at that location would be the same as the last event that occurred there. For such a routine it makes little sense to build a map into our program as the player's 'movement' is purely illusory.

8.4 Method B: seeded random movement

This method is probably the most economical on memory, which is useful for machines with only a small amount of RAM or for programs where an unusual amount of memory is required for storing data. Put simply, each time the program is run a different map will be generated, but that map remains the same, unalterable, throughout the game. The 'map' is actually a series of numbers but only an extremely able mathematician would be able to predict the map from the initial randomly chosen number. The method works as in Fig. 8.3.

```
1. USE A FORMULA TO CREATE A NUMBER

 WITH A DECIMAL POINT


2. GET RID OF THE INTEGER IN THE NUMBER


3. USE THE FORMULA TO INDICATE POSSIBLE

   DIRECTIONS


4. USE THE FORMULA TO CALCULATE EVENTS


5. NEXT MOVE
```

Figure 8.3

Because the formula is the same and the sequence of numbers is the same throughout, the result of the formula will be the same at a particular 'location' every time it is run.

The routine below outlines the way to do this. The key line is line 400 which defines function p. This function uses the current x, y, and z coordinates of the player (i.e., his or her location in the game) to calculate a decimal value. Every time the coordinates are the same the value will be the same. PROCcalc compares that value with an arbitrarily chosen constant, e.g., 0.35. If p is greater than this constant then there is an exit in that direction. If not, there is none. The variable move is used to record whether a particular direction is possible or not, being set to 1 if the exit is clear and 2 otherwise. PROCmovement takes an input direction, checks that it is possible given the current values of x,y, and z, and if it is calls PROCgo which prints the exits available from the new location by temporarily resetting x,y, and z to see if the numbers that would result are greater than 0.35. Each of the digits in the decimal part of the number derived from the function could also be used to control different events in addition to the available exits. For example, one digit could be used to control selection of the main event to be found in that location.

```
99  REM Seeded Random Movement
100 DEF PROCmovement
110 move = 0
120 IF move = 2 THEN move=0:
    PRINT "Solid rock":ENDPROC
130 REPEAT
140  PRINT TAB(0,10);"Which way?"
150  INPUT dir$:dir$=LEFT$(dir$)
160 UNTIL INSTR("NSEWUD",dir$) <> 0
170 IF dir$ = "N" THEN x=x-1: PROCgo:
    IF move=2 THEN GOTO 120
180 IF dir$ = "S" THEN x=x+1: PROCgo:
    IF move=2 THEN GOTO 120
190 IF dir$ = "E" THEN y=y+1: PROCgo:
    IF move=2 THEN GOTO 120
200 IF dir$ = "W" THEN y=y-1: PROCgo:
    IF move=2 THEN GOTO 120
210 IF dir$ = "U" AND z=1 THEN PRINT
    "You´re on ground level!": GOTO 120
220 IF dir$ = "U" THEN z=z-1: PROCgo:
    IF move=2 THEN GOTO 120
230 IF dir$ = "D" THEN z=z+1: PROCgo:
    IF move=2 THEN GOTO 120
240 ENDPROC
250 DEF PROCgo
260 PROCcalc: IF move = 2 THEN ENDPROC
270 PRINT "Exits are: ";: move = 0
280 x=x-1 : PROCcalc: IF move = 1
    THEN PRINT "North"
290 move = 0 : x=x+2 : PROCcalc: x=x-1 :
    IF move = 1 THEN PRINT "South"
300 move = 0 : y=y-1 : PROCcalc:
    IF move = 1 THEN PRINT "West"
310 move = 0 : y=y+2 : PROCcalc: y=y-1 :
    IF move = 1 THEN PRINT "East"
320 move = 0 : z=z+1 : PROCcalc: z=z-1 :
    IF move = 1 THEN PRINT "Down"
330 IF z>1 THEN move = 0 : z=z-1 : PROCcalc:
    z=z+1 : IF move = 1 THEN PRINT "Up"
340 PROCcalc
350 ENDPROC
360 DEF PROCcalc
370 p=FNp: IF p<.35 THEN move = 2 : ENDPROC
380 move = 1
390 ENDPROC
400 DEF FNp=SQR(x*x+y*y*z)-INT(SQR(x*x+y*y*z))
```

8.5 Method C: the HIMEM map

A short routine using the machine code provision of BASIC (i.e., the indirection operators described in Chapter 39 of the BBC manual and Chapter 23 of the Electron manual) can be used to store a map as a series of bytes and another routine used to recall the set of bytes around the current player location to control movement and print environment. This technique can be used for simple or more complex purposes but the principle remains the same. Firstly, it is necessary to reserve sufficient memory for storing the machine code map. On the BBC/Electron this can be done by resetting HIMEM, and similar relocatable pointers for the highest location of user RAM exists for most systems (e.g., Ramtop on the Spectrum). The reserved memory must be at least as large as the total number of locations in the map.

Into this reserved area is POKEd a series of bytes, each representing one location in the map. That is to say, for each of the reserved locations in turn we use the formula:

?(memorylocation) = room number byte

These bytes can then be used as the code for description of the player's current location, or even for direct visual mapping. Let us take the latter first. Suppose the map is a series of rooms arranged within a matrix of 12 × 12 possible locations. Within these possible 144 locations, 48 are rooms and 96 are blank walls. Each of the 48 rooms will be given an identifying number and the blank walls 0, so that the whole map if drawn looks something like Fig. 8.4. After each movement the player can be shown the current position on the map by displaying the eight surrounding locations together with the current location. This obviously works well in conjunction with the 'graphic keypad' method of controlling movement.

If any of the surrounding locations are 0 they will be displayed as a wall character, such as a graphics block from the BBC's Teletext set. (Electron users will have to use a character such as the asterisk or define their own character.) If greater than 0 the display will be a blank space. For example, if the player is at location 9,11 in the matrix in Fig. 8.4, the map of his surroundings would be Fig. 8.5.

Each time the player moves not only will the map be updated but a test will be made beforehand to see if the player can move to that next location, i.e., testing to see if the next byte (location) is greater than 0. To use such a matrix we would need to reserve 144 addresses and POKE (using the ? operator) into each in turn the appropriate value. The first 12 addresses will be locations 1,1 to 12,1 on our map;

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	1	0	0	0	0	0	0	0	0	0
2	0	0	2	0	0	0	0	0	0	0	0	0
3	0	12	3	4	5	24	0	0	0	0	0	0
4	0	13	0	6	0	23	25	26	32	44	45	0
5	0	14	0	7	21	22	0	0	33	0	0	0
6	0	15	16	8	0	0	0	0	34	0	0	0
7	0	0	17	0	0	28	37	36	35	0	0	0
8	0	0	18	19	20	27	0	0	38	46	47	0
9	0	0	0	0	0	29	0	0	39	0	0	0
10	0	0	0	0	0	30	0	0	40	0	0	0
11	0	0	0	0	0	31	43	42	41	48	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8.4

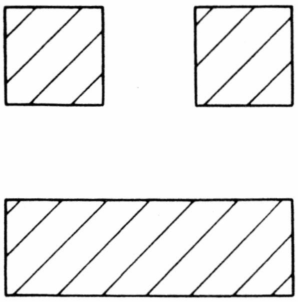


Figure 8.5

the next 12 addresses will be 1,2 to 12,2; and so on. Thus if the first address is 20 000 (decimal), we would POKE into it 0, and similarly for the next address (20 001). However, 20 002 would be given the value 1. An easy way to achieve such a series of POKEs is as follows:

```
500 P= &1FFF: REM ONE LESS THAN THE START
ADDRESS (HEXADECIMAL)
```

```
10 FOR I= 1 TO 144: REM THE NUMBER OF BYTES
520  READ A
530  ?(P+I) = A
540  NEXT I
550  DATA 0,0,1,0,0,0,0,0,0,0,0,0
560  DATA 0,0,2,0,0,0,0,0,0,0,0,0
570  DATA 0,12,3,4,5,24,0,0,0,0,0
etc.
```

To read a map like this in checking the player's movements and printing the map, we simply PEEK the nine relevant bytes, again using the ? operator. If x is the horizontal position of the map and y the vertical, then the locations around him will be x−1,y−1; x,y−1; x+1,y−1; x−1,y; x+1,y; x−1,y+1; x,y+1; x+1,y+1. However, the map is kept in memory not as a matrix but as a sequence. Each x is 1. Each y is 12. Therefore to find the desired byte we must make it x+(y*12). For example, if the player is located at point 3,2 in the matrix then x=3 and y=2, so we need to read the byte located at 20 000 plus 3 plus 2*12=address 20027. The bytes suropunding the position are thus 13 less, 12 less, 11 less, 1 less, 1 more, 11 more, 12 more, and 13 more. We can draw the nine necessary bytes using the following routine:

```
600 P=(&2000 +x +(y*12))
610 Q= ?(P−13) : IF Q=0 THEN PRINT AT (relevant screen location) (block graphic)
620 LET Q= ?(P−12) : IF q=0 THEN PRINT AT (next screen location) (block graphic)
etc.
```

Can this be reduced to a formula? Yes it can, using FOR . . . NEXT loops, as below:

```
10 REM m = FIRST ADDRESS OF MAP LOCATIONS
20 REM a AND b ARE PRINT COORDINATES
30 REM x IS WEST TO EAST
40 REM y IS NORTH TO SOUTH
50 a=2
60 b=2
70 FOR y=-12 TO 12 STEP 12
80 FOR x=-1 TO 1 STEP 1
90 p = ?(m+x+y): REM LOOK AT ADDRESS OF LOCATIONS
100 IF p=0 THEN PRINT TAB(a,b);"*"
    ELSE PRINT TAB(a,b);" "
120 b=b+1  : REM MOVE TO NEXT PRINT POSITION
```

```
130 NEXT x
140 a=a+1  : REM MOVE TO NEXT PRINT LINE
150 NEXT y
```

A similar easy routine can be used to test for 0, to check that the player is not trying to walk through a brick wall (some adventures make players think this is quite a sensible action!). The main advantage of this method is that no complex checking of screen memory is needed. The BBC and Electron organize their map of the screen in a somewhat peculiar way, which is difficult to process in a program. By using our own map of what is on the screen we do not need to look at the micro's own screen map at all.

8.6 Filling in the map

Having designed our puzzle map, placed it in memory, and written routines for examining it to display and act on what is there, we need to know what each location means. With only 68 locations it is possible to have a small routine for each location, but it is easier to write rather less. So we will have three types of routine which are called by the unique map location. These can be:

1. Simply a display routine with a simple description
2. A display routine, a simple description, and a routine which is also used by other locations
3. A display routine, a simple description, and a routine unique to this particular location

So each of the 68 subroutines will involve a description. Some will have additional routines and some of the additional routines will be general, while others will be specific to the unique location.

It is at this point that creativity, imagination, humour, mind maps, and fun come into the design. Each of the 68 places will need a description, which can be as simple as 'a tunnel' or as complex as you like. Each location and each element in the logical puzzle will have to be filled with a meaningful description. The way to work is to produce a long list of clever ideas, two or three times the amount you will need, and then select the ones that fit together best for the kind of game it is going to be. You may already have some ideas based on the general setting of the game, but now they have to be turned into specific words.

One way to do this is to draw up a list with three columns, the letters of the alphabet (used for easy reference), the objects, and what the object is needed for. A simple example is that object J might be 'a key' and elsewhere in the puzzle will be 'a locked door'.

However, it is best if most of the relationships are not as straightforward as this. In fact, many puzzle adventures make a point of being as esoteric as possible, often using puns or long trains of thought to make the thing work. For example, K might be 'a duck' which might be needed at 'a steep cliff'. What is the relationship? In order to go further you have to descend the cliff. How do you do that? It is easy—you get down off the duck. (Not a very good joke perhaps but it can really test the intelligence of the player.)

Once we have a complete list of such relationships we have all the basics of our game. It is a good idea if some sort of theme links them together, but it is not necessary. You can see in The Opal Lily that some of the puzzles are thematically linked and others are more or less arbitrary.

It is then mainly a question of writing and coding each of the separate location routines. This can require some thought.

Before coding we write a list of all the objects, which will be our noun list, and the action or actions that can be performed with each object. In addition we need a list of all the descriptions of the different locations. At each map location two types of routine will be used, one which prints the description including any objects there and one which accepts input and gives conditional responses. The description routine will thus have two components, but the player should not be able to separate them. One part will simply print the description of the location. The player will not be able to manipulate or respond to that output in detail. The other part will mention any objects there or, if the objects are hidden, a clue as to the presence of such an object.

In essence anything that can hold a person can be regarded as a location with its own description. Some commonly used places are listed in Fig. 8.6. The description can be as long and involved as memory allows. The larger it is the more the player will need to interpret, but a description which is too long without any possible player interaction will only serve to annoy. The locations should be as interesting as possible.

Try to make the relationships between places of some interest, rather than the straightforward 'you enter another room'. Characters can travel into and out of buildings, up and down hills and cliffs, across ravines and rivers, into secret passages, under bridges, etc. Where possible, extra puzzles can be set by making the entrance to a particular location problematic even if it does not depend on an object to be found. A simple example is to give in the description a choice of routes, only one of which is correct, with the others ending in sudden death, as in describing two treacherous paths down a cliff face, one of which will crumble away. More intricate can be locations which are unreachable unless the correct command is used. For example, to cross a stream the player might try to JUMP, WADE, LEAP, PADDLE, and CROSS before he thinks of SWIM.

FIELD	FOREST	MOUNTAIN	BRIDGE	HOUSE	
MANSION	CASTLE	CAVE	TUNNEL	ROOM	CHEST
WARDROBE	CLOCK	ROCKET	VALLEY	HOLE	PIT
BOX	DESK	CAR	CART	CARRIAGE	TREE
FRIDGE	CUPBOARD	WINDOW	ATTIC	CELLAR	
ORCHARD	BARREL	CHIMNEY	BOAT	RAFT	ROOF
PLANE	LADDER	STAIRCASE	LIFT	ALLEY	ROAD
VOLCANO	GLACIER	PYRAMID	DESERT	SWAMP	
SHRUBBERY	POOL	LAKE	GLACIER	ISLAND	
LAGOON	HILL	THEATRE	LEDGE	CLEFT	

Figure 8.6

In writing descriptions of locations we should also make them appear relevant to the objects which are originally located there. This means they make sense to the player (which might not matter if we are designing a nonsense or absurd game such as one based on Alice in Wonderland) and that the important aspect of each location may not at first be apparent, thus adding more problems for the player.

8.7 Creativity

At this point we will pause for a moment and consider a program which has one of the elements of artificial intelligence, namely, a degree of creativity. The program produces new ideas just as a human being might (e.g., in generating the mind maps suggested in Chapter 2). However, the program does not evaluate any of the ideas. To do so it would require a huge database of knowledge on what was practical or desirable in the real world. Creativity in human terms is a question of finding new links between existing items, a process sometimes called 'bisociation', and that is what this program does. The user has to evaluate the ideas that result, deciding which might be usefully developed.

A simple version of the program is Bisoc, given in Fig. 8.7. This simply takes any two lines of program and READs in a DATA statement from each line. The result is sometimes striking but often nonsensical and rather limited. A more sophisticated, more intelligent, or 'creative' program is one that has some of the meaningfulness of language built in so that its ideas are a little more like human ideas. In the jargon, we want a program with some 'semantic knowledge'. Ideas in Fig. 8.8 gives an idea of such a program.

```
 10REPEAT
 20R=INT(RND(90)*10):S=INT(RND(90)*10)
 30RESTORE(R+90)
 40READA$
 50RESTORE (S+90)
 60READ B$
 70PRINTA$;" ";B$
 80UNTIL FALSE
 90DATA CITY
100DATA FISH
110DATA UNDERWATER
120DATA ICE
130DATA SHIP
140DATA FOREST
150DATA FEMALE
160DATA ROCK
170DATA LEDGE
180DATA EGG
190DATA CAVE
200DATA GIANT
210DATA FISHING
220DATA GHOUL
230DATA WAR
240DATA HEAVY
250DATA GOLD
260DATA DRAGON
270DATA WOOD
280DATA ROAD
290DATA LANE
300DATA TOWN
310DATA VILLAGE
320DATA ROOM
330DATA ROUND
340DATA SQUARE
350DATA ANCIENT
360DATA WATER
370DATA FIRE
380DATA GLOWING
390DATA RIVER
400DATA GLACIER
410DATA STEAM
420DATA ERD
430DATA RED
440DATA STREAM
450DATA SCREAMING
460DATA BROKEN
470DATA WINDY
480DATA TREE
490DATA HOLE
500DATA SILENT
510DATA WHITE
520DATA ANIMAL
530DATA GREEN
540DATA FIELD
550DATA CAVERN
560DATA BOX
570DATA CUPBOARD
580DATA SWORD
590DATA CLOUD
600DATA AIR
610DATA SKY
620DATA WINGED
630DATA ROOF
640DATA HOLLOW
650DATA KITCHEN
660DATA OGRE
670DATA PATH
680DATA DELL
690DATA VALLEY
700DATA VOLCANO
710DATA SPACESHIP
720DATA LADDER
730DATA CLIFF
740DATA TUNNEL
750DATA VAULT
760DATA FLYING
770DATA CASTLE
780DATA PALACE
790DATA JUNGLE
800DATA REEF
810DATA CROWN
820DATA CORAL
830DATA AIR
840DATA CLOUD
```

```
850DATA FAN
860DATA GAS
870DATA CREEK
880DATA COVE
890DATA POOL
900DATA BAY
910DATA MOUTH
920DATA LAKE
930DATA TARN
940DATA POND
950DATA WELL
960DATA PUMP
970DATA DAM
980DATA MARSH
990DATA SWAMP
```

Figure 8.7 Bisociation.

```
  5 CLS
 10 PROCinit
 20  REPEAT
 30    PROClength("the")
 40    PRINT "THE ";
 50    PROCadj
 60    PROClength("the")
 70    PRINT "THE ";
 80   PROCchoose
 90    PROCnoun
100    UNTIL FALSE
110  END
120
130
140  DEFPROCadj
150 PROCchoose
160  R=FNR(4)-1
170  IF R<2THEN PROCnoun:ENDPROC
180  FOR I=2TOR
190   REPEAT
200      S=FNR(adjs)
210      UNTIL AA$(S,2)=NA$(T,2) OR AA$(S,2)="2"
220    PROClength(AA$(S,1))
230    PRINT AA$(S,1);" ";
240    NEXT I
250  PROCnoun
260 ENDPROC
270
280
290 DEFPROCnoun
300  PROClength(NA$(T,1))
310  PRINT NA$(T,1);" ";
320  R=FNR(3)
330  IF FLAG <R THEN PROCverb:ENDPROC
340   PRINT CHR$(8);"."
350  PRINT
360  FLAG =0
370  ENDPROC
380
390
400  DEFPROCverb
410  REPEAT
```

Figure 8.8 'Ideas' (*continues*)

```
420     U=FNR(verbs)
430     UNTIL VA$(U,2)=NA$(T,2) OR VA$(U,2)="2"
440    PROClength(VA$(U,1))
450   PRINTVA$(U,1); " ";
460   FLAG = FLAG +1
470   ENDPROC
480
490
500   DEFPROCchoose
510   REPEAT
520     T=FNR(nouns)
530     VV=0
535   REM CHECK VA AND NA AGREE - RETURNS VV =1
      IF NOT
540      IF FLAG <> 0 THEN PROCagree
550     UNTIL  VV<>1
560  ENDPROC
570
580
590  REM Initialisation
600   DEFPROCinit
610   nouns = 319
620   adjs = 64
630   verbs = 52
640   DIM NA$(nouns,2),AA$(adjs,2),VA$(verbs,3)
650   DEF FNR(X)= INT(RND(1)*X)+1
660   FLAG=0
670   FOR I=1 TO nouns
680     READ NA$(I,1), NA$(I,2)
690     NEXTI
700   FOR I=1 TO adjs
710     READ AA$(I,1), AA$(I,2)
720     NEXT I
730   FOR I=1 TO verbs
740     READ VA$(I,1), VA$(I,2), VA$(I,3)
750     NEXT I
760  ENDPROC
770
780
790  DEFPROClength(word$)
800   l=LEN(word$)
810   IF POS + l > 39 THEN PRINTCHR$(13)
820   ENDPROC
830
840  REM CHECK N+V AGREEMENT
850   DEFPROCagree
860   IF VA$(U,3)="2" THEN ENDPROC
870    IF NA$(T,2)=VA$(U,3) THEN ENDPROC
880   VV=1
890   ENDPROC
900
910
920   DATA HOUSE,0,DRAGON,1,TROLL,1,
      GOBLIN,1,KEY,0
```

```
 930   DATA STONE,0,BOOK,0,SCROLL,0,POTION,0,
       SLAVE,1
 940   DATA HAT,0,COAT,0,CLOAK,0,BOOTS,0,
       BAG,0
 950   DATA GAUNTLETS,0,ROBE,0,HELMET,0,
       CROWN,0
 960   DATA EARRINGS,0,RING,0,NECKLACE,0,CHAIN,0
 970   DATA SCARF,0,SHAWL,0,ARMLET,0,BRACELET,0
 980   DATA TORQUE,0,JACKET,0,TUNIC,0,TROUSERS,0
 990   DATA BELT,0,POUCH,0,SACK,0,SANDALS,0,
       ANKLET,0,DRESS,0
1000   DATA HELM,0,CHAINMAIL,0,ARMOUR,0,GREAVES,0
1010   DATA VANBRACES,0,BREASTPLATE,0,SHIELD,0,
       BUCKLER,0,SPURS,0
1020   DATA DAGGER,0,SWORD,1,MACE,0,HAMMER,0
1030   DATA FLAIL,0,SPEAR,0,JAVELIN,0,
       MORNING STAR,0
1040   DATA BOW,0,ARROW,0,SLING,0,CROSSBOW,0,
       PIKE,0,LANCE,0,DART,0
1050   DATA WEAPON,0,CLUB,0,STAFF,0,
       WAND,1,SCIMITAR,0
1060   DATA TRIDENT,0,HALBERD,0,AXE,0,PICKAXE,0,
       JERKIN,0,WHIP,0,QUARREL,0
1070   DATA SCABBARD,0,QUIVER,0,CAP,0,WIG,0,
       GLOVES,0
1080   DATA WINESKIN,0,TINDERBOX,0,TORCH,0,
       SPIKE,0,BOAT,0,POLE,0
1090   DATA MIRROR,1,LANTERN,0,CHEST,0,MAP,0,
       SCROLLCASE,0
1100   DATA CANDLE,0,PACK,0,BEADS,0,CROSS,0,
       FLASK,0,SADDLE,0,BLANKET,0
1110   DATA CART,0,HERB,0,TREE,1,FOOD,0,BEER,0,
       BREAD,0,APPLE,0,WINE,0
1120   DATA MEAT,0,CHICKEN,1,COW,1,DOG,1,
       DONKEY,1,GOAT,1,HAWK,1
1130   DATA HORSE,1,MULE,1,OX,1,PIGEON,1,PIG,1
1140   DATA SHEEP,1,BIRD,1,EAGLE,1,SWAN,1,
       SQUIRREL,1,SPARROW,1
1150   DATA DUCK,1,FERRET,1,CAT,1,BEAR,1,
       WOLF,1,FOX,1,LION,1,TIGER,1
1160   DATA APE,1,ANT,1,BADGER,1,BANDIT,1,
       BANSHEE,1
1170   DATA BASILISK,1,BEAVER,1,BEETLE,1,
       BERSERKER,1,DRAGON,1,BOAR,1,
       BROWNIE,1
1180   DATA BUGBEAR,1,BULL,1
1190   DATA CAVEMAN,1,CENTAUR,1,CENTIPEDE,1,
       CHIMERA,1,COCKATRICE,1,CRAB,1
1200   DATA CROCODILE,1,DEMON,1,DEVIL,1,
       DINOSAUR,1,GENIE,1,DRYAD,1,DWARF,1
1210   DATA ELEMENTAL,1,MAN,1,EEL,1,ELEPHANT,1,
       ELF,1,WOMAN,1,GIRL,1,BOY,1,CHILD,1
1220   DATA BABY,1,FAIRY,1,FAWN,1,DEER,1,FROG,1,
       STAG,1,FUNGUS,1,GARGOYLE,1
```

Figure 8.8 (*continues*)

```
1230  DATA GHOST,1,GHOUL,1,GIANT,1,GNOLL,1,
      GNOME,1,GOBLIN,1,GOLEM,1,
      GORGON,1,SLIME,1,GRIFFON,1,
      HALFLING,1,HARPY,1,HOUND,1,
      HOBGOBLIN,1,HAWK,1,HYDRA,1
1240  DATA OGRE,1,KOBOLD,1,INSECT,1,LEOPARD,1,
      LICH,1,LIZARD,1,LIZARDMAN,1,LURKER,1,
      WEREBEAR,1,WEREWOLF,1,WERERAT,1
1250  DATA MANTICORE,1,MEDUSA,1,MERMAN,1,
      MERMAID,1,MINOTAUR,1,MOULD,1,MOLE,1,
      MUMMY,1,NAGA,1,
      SPIRIT,1,NIXIE,1,NOMAD,1,NYMPH,1
 1260 DATA ORC,1,MAGE,1,OWL,1,PEGASUS,1,
      PIERCER,1,PIRATE,1,PIXY,1,MAGE,1,
      SORCERER,1,RAT,1,ROCK,0,MONSTER,1,
      SALAMANDER,1,SATYR,1
 1270 DATA SCORPION,1,SHADOW,1,SKELETON,1,SLUG,1,
      SNAKE,1,SPECTRE,1,SPIDER,1,SPRITE,1,
      STIRGE,1,SYLPH,1,TOAD,1
 1280 DATA ENT,1,OAK,1,TROLL,1,SOLDIER,1,
      UNICORN,1,VAMPIRE,1,WASP,1,BUSH,0,
      FERRET,1,WIGHT,1,WRAITH,1,WYVERN,1,
      ZOMBIE,1
 1290 DATA RIVER,0,SEA,0,POND,0,LAKE,0,
      WATERFALL,0,ISLAND,0,FORD,0,STREAM,0,
      OCEAN,0,MOUNTAIN,0,HILL,0,STONE,0,
      CLIFF,0,RAVINE,0
 1300 DATA CITADEL,0,PALACE,0,CITY,0,CAVE,0,
      VILLAGE,0,FARM,0,FORTRESS,0,FOREST,0,
      TOWER,0,HUT,0,CASTLE,0
 1310 DATA GLADE,0,WOOD,0,TOWN,0,GLACIER,0,
      RIDGE,0
 1320 DATA THIEF,1,WIZARD,1,KING,1,QUEEN,1,
      PRINCE,1,KNIGHT,1,SORCERER,1,FIGHTER,1,
      CLERIC,1,DRUID,1,PALADIN,1,ASSASSIN,1
 1330 DATA SORCERESS,1,NURSE,1,PRINCESS,1,
      RANGER,1,BARD,1,INNKEEPER,1,SMITH,1,
      COOPER,1,DOCTOR,1,MIDWIFE,1,WITCH,1
 1340 DATA MERCHANT,1,FARMER,1,WOODSMAN,1,
      HUNTER,1,FURRIER,1,BOOK,0,AMBASSADOR,1
 1350 DATA COOK,1,SAILOR,1,GUARD,1,
      MAN AT ARMS,1,OSTLER,1,GROOM,1,PEASANT,1
 1360 DATA COLD,2,STUPID,1,STEEL,0,GOLD,2,
      COPPER,0
 1370 DATA BLACK,2,BLUE,2,WHITE,2,GREEN,2,
      RED,2,ORANGE,2,PURPLE,2,SILVER,2,BRONZE,0
      YELLOW,2,
 1380 DATA INTELLIGENT,2,CLEVER,2,IRON,2,WOODEN,0,
      RUBY,0,SAPHIRE,0,TOPAZ,0,JET,0,IVORY,0,
      MOONSTONE,0,OPAL,0
 1390 DATA AMETHYST,0,AGATE,0,GARNET,0,
      TURQUOISE,0,CRAZY,1,LAZY,1,SILLY,1,
      BROKEN,0,HOT,2,ICY,2,FIERY,2,UNDEAD,1
 1400 DATA GRANITE,0,HARD,2,SOFT,2,GENTLE,1,
     STERN,1,COMFORTABLE,0,STRONG,1,WEAK,1,
     UGLY,2,ATTRACTIVE,2
1410 DATA WOUNDED,1,HEALTHY,1,ILL,1,MAD,1,
     PERFECT,2,ANCIENT,2,DAMAGED,0,
     FRIGHTENING,2,WORRIED,1
1420 DATA HOLY,2,WICKED,1,EVIL,2,FURTIVE,1,
     DRUNKEN,1,HUNGRY,1,DECEITFUL,1
1430 DATA KEPT IN,0,0,FOUND NEAR,2,0,
     MADE BY,0,1,MAKER OF,1,0,ENEMY OF,1,1
     POSSESSED BY,0,1,POISONED BY,1,1,
1440 DATA ENEMY OF,1,1, HATED BY, 2,1,
     PURSUING,1,1, CONCEALING, 2,2
1450 DATA MAKER OF,1,0, MADE BY,0,1,
     OWNED BY,0,1, LIVING BY,2,1
1460 DATA AFRAID OF,1,2, CURSED BY, 2,1
1470 DATA WORRIED BY,1,2,KILLED BY,1,2,
     IN LOVE WITH,1,1,HATED BY,2,1,OWNED BY,2,1,
     GIVEN TO,0,1,TAKEN FROM,2,1
1480 DATA CHASED BY,1,1,DISCOVERED BY,2,1,
     HIDDEN BY,2,2,FRIEND OF,1,1,
     ASSASSIN OF,1,1,RULER OF,1,2,HOME OF,0,1
1490 DATA LIVING WITH,1,2,SLAVE OF,1,1,
     HIDING FROM,1,1,RUNNING FROM,1,1,
     RUNNING TO,1,2,LOVED BY,2,1,STOLEN FROM,2,2
1500 DATA CAUGHT BY,1,1,CAUGHT WITH,1,2,
     SERVANT OF,1,1,MASTER OF,1,2
1510 DATA VALUED BY,2,1,LOST NEAR,2,0,
     FRIEND OF,1,1,CONQUERER OF,1,1,
1520 DATA ENCHANTED BY,2,2,BEARER OF,1,0
     PROTECTOR OF,1,2,WORSHIPPED BY,2,1,
     FOUND NEAR,2,2,LOST BY,2,1
```

Figure 8.8 'Ideas'

The value of a program like this depends on its database. This particular idea generator produces ideas to use in adventure or fantasy role-playing games. It attempts to create new fantasy ideas, but different databases could be used for other types of plot or scenario, such as Science Fiction or a Western. Other databases, rather more remote from the type usually found in an adventure game, could include different kinds of human social or political relationships, or the kinds of relations typical of TV soap opera, or any situation where two elements are linked in some way. In principle any simple domain in which ideas can be generated more or less 'mechanically' by linking two hitherto remote concepts could be incorporated into a program such as this. It is therefore included here for three reasons. Firstly, it gives just a glimpse of what creativity in AI might involve. Secondly, you can use it to produce original ideas for your own games and adventures. Finally, you might like to adapt it to produce ideas in an area that you are particularly interested in. For example, you might want to write a story or a play—create a database of plot elements and a program like this can link them to give you new scripts. Or you may want to invent a new meal or a drink—create a database of ingredients and this will do the job for you. You might even be able to adapt it to suggest solutions to difficult problems by giving it a database of the component elements in the problem and the desirable goals, and evaluate the possible combinations it gives.

You can probably see from this that any real ‘creativity’ here comes from the user rather than the program. The user creates the database and the user evaluates the results. It is the user who interprets the program’s output and if the user cannot apply the output to the desired task then no creativity has occurred at all. The program is therefore really a tool for creating new phrases and new links between words. It is the user who finds the usefulness (or not) of the ideas behind those words. This is actually quite close to part of the processes of some human creativity. Very often people will have ideas which they change, alter, mature, and evaluate before using them, and the original source of the ‘idea’ can be very remote from the final solution. Thus the program really serves to spark the imagination rather than to replace it. It throws up unlimited suggestions in the hope that sooner or later something will occur that the user can make sense of.

The principle could probably be extended. One could imagine a program which operated a number of the ‘rules’ for lateral thinking popularized by psychologists like Edward de Bono. Much of his work shows that people fail to find solutions to problems because they get trapped in particular patterns of thinking and find it impossible to enter into a different mode of thought. If we had a program which, as a matter of course, simply applied all the possible strategies to a particular idea this should create a number of suggestions which, however absurd, suggest to the user more sensible variants that break the restricting thought patterns in which the problem solver is trapped. (If you can write such a program then you are probably made for life!)

The ideas program works by building up noun phrases concerning objects/creatures/beings and then linking two or more such phrases through a verbal relation of some kind. To avoid ungrammatical constructions certain limitations are built into the database and the program, such as using ‘THE’ rather than ‘A’ to avoid the problems of agreement. However, most of the possible relations are semantically correct, though some might seem a little odd. The basis of this correctness is the number held in the DATA statements after each word or phrase. All nouns and adjectives have a single-number code; all verb phrases have two-number codes. Code 0 means that the word is inanimate, 1 means that it is animate, and 2 means that it could be either. The verb phrases have two codes because they are preceded by a word that may be animate or inanimate as well as followed by the same choice. So the phrase 'hated by' must be followed by an animate phrase because only animate things can hate, but may be preceded by either an animate or an inanimate phrase as both classes of thing can be hated. Its code is therefore '2,1'.

The program simply selects a noun phrase, then a verb phrase, and then another noun phrase. However, each noun phrase can have up to three adjectives and a noun phrase may itself consist of an embedded verb phrase plus a noun phrase. So a simple selection would be a simple noun phrase plus a verb phrase plus another simple noun phrase, all of them agreeing and none using adjectives, such as:

THE PRINCESS +IN LOVE WITH + THE FROG

The next most complex version adds one or more adjectives, such as:

THE GIANT MUSHROOM BROKEN BY THE UGLY
BLACK GOBLIN

Finally, the most complex is one which modifies the final phrase by adding a verb phrase to it, such as:

THE STONE HELMET WORN BY THE GREEN
DRAGON FEARED BY THE ANCIENT CLEVER WIZARD

Of course, for every idea that makes reasonable sense you will still find quite a number which are nonsensical, funny, or impossible. If you wanted to reduce these you would have to build in other types of semantic constraints in addition to the test for animacy. The kinds of things you might test for could be size, mobility, whether they can be eaten or not, whether they have emotions, etc. There is a huge number of possible relations and every one you add to the program will reduce the inventiveness of the ideas and will have some exceptions which make the rule difficult to apply. Perhaps you can begin to see why making a computer intelligent is such a difficult task. Just to make it understand the word 'SACK', for example, means that it has to know a sack is floppy unless something is in it, that it can be moved, but only if an animate thing is moving it (unless the inanimate thing is actually a force such as the wind or the tide), that it is a container, that it is not a house in the human sense but that things can live in it, that it can be cut and torn, that it can leak but not hold liquids , . . . , and so on. You can imagine how difficult it would be to program all the knowledge required to use language properly even in a very limited area. Nevertheless it is fun trying.

8.8 The creative game

It should be clear by now that there are really only two types of game, particularly when we think about adventures. On the one hand there is the puzzle type adventure which is a real test of the player's intelligence and imagination, but it has the major defect of being fixed in structure and content, so that once it has been solved it will not be played again. On the other hand, there is the random adventure, the dungeons and dragons combat game. Rather than being fixed at the outset, this is fundamentally a series of random structures. The player has relatively little to do in the way of developing a consistent and intelligent strategy, and little chance of predicting how the parts of the adventure might fit together. Monsters and treasures are randomly distributed and alterations in the character's variables, abilities, and characteristics are more or less accidental (depending on which locations were gone to and when). Therefore though each encounter may be interesting, its fundamentally random nature will eventually lead to boredom.

One approach which may eventually overcome this problem is to devise a game which structures itself so that it is different each time it is played yet has a logical structure, thus testing player intelligence and developing strategies, and which changes during play to respond to the player's previous decisions not randomly but intelligently. Artificial intelligence can provide ideas and techniques on how to go about achieving this. As usual, however, we run up against the problem which the current generation of popular microcomputers have—insufficient memory for large artificial intelligence components—so at present we can only investigate the possibilities and try to find new ways of development.

However, it should be self-evident that most of the decisions a programmer makes in designing an adventure could be handed over to a computer, provided we could specifiy the decisions clearly enough for a workable algorithm to be produced. Two approaches are possible. One would be to design a suite of programs which, like the programming aids described in Chapter 9, successively produce stages of the adventure until the final product is a complete block of code which, when run, is an adventure. The first program would generate a series of ideas, a plot, or a set of puzzles, using a method something like that used in the Ideas program. A second program would then fit the puzzles and ideas together to make a map, i.e., a linked series of ideas. A third program would look at each described location in the map and turn it into a full description by using phrases like 'You see a . . .', 'Ahead there is a . . .', 'After a long journey you enter . . .', and so on. Each location where a problem was to be solved would be marked, just as in planning an adventure a designer would note which areas needed fuller development, and a subsequent program would code each of these markers to represent one of a series of puzzles developed by another program. You can probably see that if 'creating an adventure' is broken down into its component stages it would not be too difficult to construct such a series of programs. The key program, the most intelligent program, would have to be the creative or 'imaginative' program that set up the design in the first place and, as we have seen, creativity is not well enough understood to be modelled accurately as yet and it may never be.

Consequently you might settle for the lesser aim of the second approach. This would be to write *one* program which reconfigured itself each time it was played. This might also seem impossible, but in principle it is not (though I have yet to find a playable implementation). The basic set-up would always be the same in some sense—let us say a fantasy city—and it would always have the same potential components (such as a list of 50 possible locations). However, each time it is run it could use a number of different procedures to set up a totally different game. First of all it could set up a street map which links all the locations together, using some constraints which make sure, for example, that the palace is not in the middle of the beggars' quarters and that the shipbuilder's yard is near the dock. This would be very easy as the program would simply have a matrix holding the maximum and minimum distances allowed between any two locations. (Here is one reason why memory becomes crucial.)

It could also generate new street names for each of the streets and a set of character names, together with personality variables which control the behaviour and nature of those characters. This would be something like the set up of a purely random game up to this point. However, one of the features of such characters would be their 'home' (i.e., their base location) and their allowed routes (i.e., the streets and locations they are allowed to enter). Add to this a routine which moves some of the characters around each turn and we have a different populated city each time it is played.

On top of this we would want a routine that sets up a scenario, i.e., a task to be accomplished. Firstly, a number of objects are scattered around which have value either to the character or the player's character or both. Then the program puts together two sets of two plot elements to create the scenario, using the objects, street names, locations, and characters as its raw data. For example, suppose we had the following list of possible plot elements:

Group 1

Kill
Capture
Find
Return

Group 2

Greybeard the Wizard
King Aelfric
Stoutfish the Dwarf
Princess Freda

Group 3

The palace
The dungeon
The Green Tower

Group 4

The diamond sword
The packet of seeds
The magic flute
The pack of hounds

The rules of combination, like those in the Ideas program, would say which combinations were allowed to string items together. For example, one possibility would be Group 1 + Group 2 "AND" + "Return" + (Group 2 or Group 4) + "TO" + (Group 3 or Group 2). This could give scenarios like:

"You must capture Greybeard the Wizard and return Greybeard the Wizard to the Green Tower"

or:

"You must kill Stoutfish the Dwarf and return the magic flute to Princess Freda"

Obviously the semantics of such combinations must be thoroughly worked out beforehand. My example would allow a scenario like "You must return Princess Freda and return Princess Freda to Princess Freda", which does not make sense. Equally obviously, this plot-devising routine would hold some internal representation of the plot it has devised in order to test that the conditions had been met. Each character would have a value, as would each location, and if the plot was "Return character x to location y" then the program would check each turn of the game to see if character x was in location y and accompanied by the player's character.

One final refinement to such a program would be to add to each character a small 'knowledge' component. Once the plot, locations, characters, homes, routes, and objects had been established it would be a simple matter to allow each character to 'know' something about the rest of the town, such as "The dwarf is in the pub" or "The packet of seeds opens the door to the princess' library" or "The wizard walks every day in Sluicegate Lane". This information could be extracted from the characters in various ways—by bribery, by fighting, by being nice to them, by giving them the required object, or by the tried and trusted method of 'bashing hell out of them'. This introduces another possibility—that the player's character and the 'monster' could interact intelligently.

To make a program behave intelligently is to make it understand. That means it has to know not only what the player has just done but also what the player has been doing for some while and what these actions mean (i.e., what the player is likely to do in the future). Normal games programs consist of a series of immediate responses, based on what the player has just done. In the puzzle game the player types in a command and the program interprets the command, then forgets it, and waits for the next one.

In the combat game the program calculates the effect of each blow, then forgets it, and waits for the next one. It calculates each combat in this way. When it is over it forgets it and waits for the next one.

An intelligent program, however, remembers what has been done, tries to interpret it (by discovering an underlying meaning or strategy), and tries to predict what might happen next. According to that prediction it will make a response which is not based on the player's last actions, but the overall trend of the actions, and which is not purely responsive and dependent, but independent, directed by the 'goals' of the program itself. This is essentially what happens in the Scissors game in Chapter 5.

Exactly the same approach can be used in adventure games. For example, let us look at the concept of friend/enemy mentioned above. A simple flowchart for understanding this concept might be as in Fig. 8.9.

```
(1) PLAYER ACTS TOWARDS MONSTER

(2) IF THE ACTION IS HOSTILE THEN LET

ENEMY = ENEMY + 1

(3) IF THE ACTION IS NOT HOSTILE THEN

LET FRIEND = FRIEND + 1

(4) IF FRIEND < ENEMY THEN FIGHT THE

CHARACTER

(5) IF FRIEND >= ENEMY THEN GIVE

CHARACTER REWARD
```

Figure 8.9 **The concept of friend/enemy**

In other words, this involves setting a variable for each of the two concepts, updating the two variables every time the player performs an action, and comparing the two variables every time a response is needed. Consequently, two possible types of routine are also needed, a friendly one and a hostile one. This could mean that for every monster in the game two variables are needed and two sets of response routines. Obviously this is likely to double the amount of memory we will need for a fixed number of encounters, though there are ways to reduce this. For example, the enemy/friend dichotomy can be held in one variable, with hostility subtracting one from the value of that variable and friendliness adding one. Thus if the number is positive the relationship is friendly; if negative it is hostile. Also the same response routines can be used with different variables for different monsters. Even so, more memory is needed for 'intelligent' routines.

For every axis of intelligence we wish to add, the cost in memory and coding will increase. If we wanted to record generosity versus miserliness, for example, or caution versus recklessness, or talkativeness versus shyness, or stupidity versus cleverness we would need to add variables and routines for each of these. While such additions will greatly enhance the game, what we are actually adding is potential. In any particular playing of the game many of these routines might never be used. A very unfriendly, generous, cautious, talkative, and stupid player would only ever use half the game. Potentially the game would be much more interactive and exciting, but some of it would not be used.

This is the problem with such programming. Intelligence involves being able to respond to a wide number of situations, so they have to be programmed in, but if a particular situation does not occur, that part of the program will not be used.

The same is true in the example of writing a game which writes itself. Suppose, for example, the game could choose from a set of possible puzzle types each time it was run. Let us say it had the choice of anagram, cypher, and square-root problem. It chooses 'cypher'. It then uses the cypher routine to produce an actual cypher and it holds this in memory as being the particular problem to solve. It then looks at all the empty locations in the game and picks one to hold that cypher. It then chooses another empty location and sets a flag which indicates "if the cypher has not been solved, this location is blocked". Then it selects from a number of possible descriptions the actual description used: "Your route is blocked by a door with a combination lock". Finally, it checks that a route is possible between the two locations.

The process just described is very much like the processes described earlier in this book in more detail, to guide you through the task of writing your own adventure. In other words, such a program would be doing what a programmer could do and doing it intelligently. However, it would involve at least eight routines which we did not incorporate in our game, plus extra memory and variables to hold the possible choices, and this would only set one type of location. If this was to be a semantic puzzle, such as one based on a pun, a different set of routines would be needed.

It therefore seems unlikely that micros such as the BBC or Electron will support such elaborate games, unless additional processors become available which expand the maximum memory. It might also be possible to reduce the problem of holding a number of potential routines which might not be used, in making characters and computer responses more 'intelligent', by using disks. If the least-used routines are held on disk they do not need to use any of the computer's RAM except when called and loaded. Thus the same RAM could be used for different routines at different times in the game. However, only expensive micros like the Apple presently have such adventures, and even these slow the response time of a game down quite noticeably.

There are other possibilities for improving existing types of game substantially even within the limits of the current generation of micros. One way to do so is to combine both the puzzle and combat types. There are one or two games which attempt to do so, but some programs which claim to do this are really several games masquerading as one.

If you understand the principles behind both types of game you should find it easy to improve them and consequently may hit on ways of integrating them. It is a relatively easy job to add combat locations to the puzzle game, for example. A particular location could be passed through not by solving a problem but by fighting a monster. That fight could be helped or hindered by the objects which the player's character is currently carrying. A sword might make him fight better; a plank might get in his way. By a device such as this an extra dimension can be added to the puzzle game as the player cannot be certain that a particular set of objects will always lead to success.

Similarly the combat game can have puzzle locations within it. Suppose the reward for destroying an enemy starship is that Igor gains its cargo. The cargo could be fuel or ammunition, in which case this simply improves the fighting abilities of the victorious ship. However, it could also be an object, without which the ship could not go to a particular location; this could be a map of a new solar system, or an electronic key to a starport, or a guidance system for use in asteroid fields.

In this way puzzles, combat strategies, and even arcade-type real-time action can be combined in one game. Variety is not only of structure and content but also of play. The player will have to make decisions of different kinds and in different combinations, testing all of his abilities and not just a limited set of them.

It is also possible to use techniques like those used in artificial intelligence to enhance a game without trying to go to the full extent of making the program cope with every eventuality. For example, in the combat game it is a comparatively simple matter to keep a record of the number of victories the player has scored over a particular kind of monster. If this record shows that the player tends to defeat one kind of monster but run away from another kind, the program can alter the game so that there are fewer of the 'easy' monsters and more of the 'hard' ones. Or it can just keep count of the monsters killed and use this as a variable which determines the strength of the next monster encountered.

Intelligence could also be built in to a routine like that described in Sec. 7.2 for talking between monster and character. For example, suppose your program keeps track of the number of goblins the player kills, and the number he tries to bribe or subject to some other tactic. This can be used as an index of 'friendliness/hostility to goblins'. Suppose the player meets a very powerful goblin and does not want to fight it. The player does not have any money, so cannot bribe it, and needs the treasure which the goblin is guarding. The only option is to talk. In the program's database will be a number of keywords to look for in the talk routine and each will have a value, such as the following:

Word	Value
Evil	−3
Nasty	−2
Ugly	−1
Nice	+1
Intelligent	+2

The talk routine, having detected such a word, adds its value to the index of hostility and friendliness. If the index drops below −9 the goblin will attack. If it goes above +9 the goblin will give the treasure away. Otherwise, providing the player does not attack, he will carry on talking, his response depending on the current level of the hostility index.

If the index is −8, the goblin might say "Clear off, you lousy elf", but if it is +8 he might say "I'm glad you came because I've been looking for someone to give my diamonds to". It would not be necessary to have a different conversational gambit for every value of the index if there was not room to store a large amount of text, and variety in output can be achieved by combining phrases. Again, these phrases would have to be valued in the range of the index (−9 to +9), but some phrases might have a wide range of values, for example:

Phrase Number	Phrase	Value for hostility of output
1	I'M GLAD YOU'VE COME	+7 TO +9
2	I HATE ELVES	−6 TO −9
3	BECAUSE	−9 TO +9
4	CLEAR OFF	−5 TO −9
5	AND	−9 TO +9
6	BUT	−9 to +9
7	I LIKE ELVES	+6 TO +9
8	YOU LOUSY ELF	−4 TO −9
9	TELL ME MORE	−7 TO +7
10	I'LL SKIN YOU ALIVE	−8 TO −9
11	I'VE SOMETHING FOR YOU	−3 TO +4
12	I'M GOING TO LET YOU HAVE IT	−5 TO +3

The phrases used at any particular time would be chosen so that they fitted together sensibly and all were allowed by the current hostility index. Even more variety can be included in such output by including variables or randomly chosen words. For example, phrase 8 could be held as "YOU" + A$(R) +N$, where A$() is an array of insulting adjectives, R is a random number, and N$ holds the kind of character that the player has chosen (wizard, elf, dwarf, etc.). By building up small elements in this way a great deal of complexity can be built into a game using relatively little memory, and this complexity will be meaningful, not random.

Do not forget that if we put a variable or a set of data in our program for one purpose we can use it for many other purposes. For example, a simple routine can be used to allow monsters to give clues to the whereabouts of treasure as the reward for victory, rather than giving out the treasure itself. Such a routine can use the data already present and be given out either as a truth or a lie.

For example, we might want the output to be "Spare my life and I'll tell you where x is". The monster might know where x is, or he might not, and as x would be one of the treasure items located somewhere in the program, no new variables would be needed for telling the truth. However, if the monster is to lie, it has to choose a treasure from the set of possible treasures and a location for it. The following routine would do the job:

```
99 REM ROUTINE FOR LYING MONSTERS
100 R = INT(RND*3) +1
110 IF R =1 THEN L = treasurelocation :
    T$ = treasurename$:GOTO 140
120 R = INT(RND*4)+1: L = INT(RND * 20)+1
130 FOR I = 1 TO R: READ T$: NEXT I
140 PRINT "If you spare my life I´ll
    tell you where the ";T$;" is."
150 REM (player spares monsters life)
160 PRINT "The ";T$;" is at ";L
180 RETURN
190 DATA silver crown, gold torque, ruby, emerald
```

Line 110 decides whether to lie or not. There is a one in three chance that the monster will tell the truth. (Of course, some monsters might be more truthful than others, in which case 3 will be replaced by the truthfulness variable.) Line 120 sets the variables T$ and L to the treasure's name and true location respectively and then sends control to the output lines.

Line 130 selects the lie, choosing one of four treasures and one of twenty locations. Line 140 reads the name of the treasure into T$, using the DATA statement at 190, which will be the same DATA statement used in originally setting up the map. Lines 150 to 180 output the information. Here seven lines of new program add a whole new dimension to the game; these could be reduced by, for example, letting the random number which determines if a lie is to be told also determine the actual lie to be told.

There are many ways of adding new aspects to a basic game by using the existing structure. Monsters can be 'disguised' as other monsters; puzzles can use data already defined in the program for other purposes; the subroutine used to determine combat can also be given different parameters or variables and used to determine if a door can be opened, if a chest explodes, if a monster is asleep, etc.; a record of generosity can be used to determine the size of bribes needed (monsters ask for more because they have heard that the player is generous), the price of equipment, the friendliness of 'good' creatures, the chance of a bribe being successful, or the effectiveness of magic (the gods reward a generous player); input insults can be 'remembered' and used later about the player; objects which are useful in one situation can be hazards in another—the possibilities are endless.

8.9 Bribery and gambling

As interaction with monsters is one of the key areas for possible intelligence in certain kinds of game it is worth considering some additions that can make it more interesting. Two such additions are bribery and gambling.

Both of these involve the same kind of exchange with probably the same result—the player loses money and so is less likely to be able to buy what is needed in other locations. Bribery consists of the monster asking for money or some other gift and if he gets what he wants the player can proceed without harm. The bribe function can be incorporated in one of two ways—either the player has a free choice of it as one of the strategies, or before the player is allowed to make a choice the monster will ask for a bribe.

A way of constructing the routine to fit our game is given in Fig. 8.10. The variable 'money' represents the current total of silver pieces that the character has. The routine can be improved by making the monster ask for weapons or other items that the character is carrying. This could be the fall-back request if the character has not enough money or it could be randomly determined before the routine is run. Alternatively, the monster could steal all of the player's money if the decision is made not to bribe.

Gambling is slightly different in so far as it must be possible for the player to increase his or her money. The gambling could, of course, be for something other than money, such as armour or even the character's life, but in all cases there should be the possibility that the character will gain something out of the encounter. As gambling contains its own risk there is no need to combine it with any other type of routine, so it could be a totally separate kind of encounter.

```
100 REPEAT
110 PRINT "Do you want to:"
120 PRINT " (F) Fight"
130 PRINT " (R) Retreat"
140 PRINT " (B) Bribe"
150 A$ = GET$
160 G = 0
170 IF A$ = "B" THEN PROCbribe : IF G = 1
THEN ENDPROC
180 IF G = 2 THEN PROCfight
190 CLS
200 UNTIL FALSE
3500 DEFPROCbribe
3510 LET R = FNR(money)
3520 PRINT "He says he will accept ";R"
silver pieces."
3530 PRINT "Will you pay?"
3540 INPUT A$
3550 IF LEFT$(A$,1) = "N" THEN ENDPROC
3560 IF money < R THEN PRINT " Try to
swindle me , would you?": LET G=2 : ENDPROC
3570 money = money-R
3580 G = 1
3590 ENDPROC
```

Figure 8.10

The key to a gambling routine is the curve of probability used. The simplest would be a straight-line graph, so that the chance of winning remained the same throughout the game, irrespective of the results of previous gambles or the amount of money gambled. This would be represented by a formula like R = RND(3): IF R = 1 THEN PRINT "You win". Here the player has one chance in three of winning every time that a bet is made.

Gambling is more interesting if there is a correlation between the amount of the possible prize and the degree of risk. We could set up a table of odds where the lower the chance of winning the higher the possible reward. Thus betting at three to one gives Igor a one in three chance of winning, but would triple his money if he won, betting at twelve to one gives only a one in twelve chance of victory but success means twelve times the reward. A simple way to build this into a routine is to ask the player for the odds required and use this in the random statement:

```
100 PRINT "How much would you like to bet?"
110 INPUT A
115 IF A > money THEN PRINT "I´m afraid your calculator
    needs new batteries": CLS: GOTO 100
120 money = money-A
125 PRINT "What would you like to multiply
    your ";A;" silver pieces by?"
130 PRINT "Please type a number
    between 2 and 20 "
140 INPUT B: IF B<2 OR B>20 THEN PRINT
    "Perhaps you should learn some maths
    before you play with the big boys?":
    CLS: GOTO 120
150 PRINT "That´s a cool ";A*B;
    " you could win"
160 R = INT(RND *B) + 1
170 IF R = 1 THEN PRINT "And you win it!!!":
    money = money + R :RETURN
180 PRINT "Oh what a terrible shame.
    You lose. Never mind, its only money"
190 IF money > 0 THEN PRINT "Another go?"
200 INPUT A$
210 IF A$(1) = "Y" THEN GOTO 100
```

Any of the micro gambling games, such as fruit machine or roulette or dice, can be adapted to fit in as a subroutine in a larger game, having the advantages of more complex and varied odds and often interesting graphics. A simple way to test the player's knowledge of probabilities is to ask him to bet on the total of two six-sided dice, giving either a flat doubling of the bet if he wins (in which case he is best advised to guess 'seven' every time) or linking the winnings to the odds. For example, throwing two dice will give an average of one 12 in every 36 throws, so we can safely offer odds of 30 to 1 on a 12 and be sure that the player will lose in the long run.

To make things slightly more difficult for the player we might make it easy to win by gambling in the early stages of the game, when the player may need all the help possible, but increasingly difficult as time goes on. The counter of time could be a real-time counter, a counter of the number of moves the player has made, or could be some measure of the current success of the player, such as the overall points score or a counter which holds the number of times a bet has previously been successful.

9 AN ADVENTURE

9.1 Writing an adventure

Having discussed the different aspects of and approaches to designing adventures, let us actually write one. The adventure described in this chapter could hardly be called intelligent although, like all good adventures, it takes intelligence to play it. However, we might happily call it 'clever' because it does a few things which make it a little human. The main reasons for designing an adventure of this nature rather than an adventure filled with artificial intelligence routines are three:

1. The Electron does not really have enough memory for good AI routines without a substantial amount of machine code. You will see that I have resorted to a number of methods to save memory in this adventure but even so it is very cramped and what AI there is happens to be rather limited. BBC users will find more elbow room if they adapt the program to run in mode 7. The additional 7K should allow substantial expansion of the language processing and other 'clever' bits in this game. Even so, for a really intelligent adventure either larger RAM or machine code is required.
2. The adventure described here is in many ways introductory in nature and could have been altered in several ways to increase the 'intelligence' it has. However, most people reading this book will never have designed an adventure at all so the design here has to be somewhat introductory.
3. Finally, I have tried to show the kinds of things that can be done rather than carry any one of them to its logical conclusion. For example, it would not be difficult to expand the text processing routine to accept a much greater range of possible English sentences, but this would have meant that something else would have been left out of the game. The purpose of this book is as much to make you think as to present you with actual routines because several of the ideas raised in this and other chapters are new to game programming and have not been explored in detail by anyone. You could be the first.

Before we get down to the business of the game design it is necessary to describe some of the methods that have been used in order to save memory. To end the book I have designed an adventure with 81 locations, with a vocabulary of 140 words, and containing over 40 objects. It accepts input which is more complex than normal two-word commands, as well as allowing two-word, single-word, and single-letter commands in some cases. It checks the spelling of the entries and makes suggestions about the command that the player might have intended to type in. It uses text compression to give reasonable descriptions of the locations. It allows the user to carry out a number of actions which are not part of the game's puzzle but which would be perfectly acceptable in the real world. It monitors the personality of the character playing so that events and other characters respond accordingly.

None of these things on their own make this an intelligent adventure, but intelligence is as much a matter of versatility as anything else. This is where human beings still have the edge on machines. We are much more versatile, can command a whole range of skills, and, though we may not be perfect in any one area, we have a range of competence and skills and can switch between them at leisure. So one major feature you should aim for in trying to make a game appear intelligent is to achieve variety. We have already considered this from the point of view of entertaining the player, but it is equally important if we want to achieve a human-like nature in our games. This adventure gains a degree of intelligence from the fact that it can cope with a number of different aspects of human behaviour, though not being especially good at any particular one of them.

All of this means that drastic attempts have been made to save memory, and this makes programming the adventure not simply a matter of typing in a listing. All adventures are programs for processing databases, and this one has a large database. However, if we were to load that database within the main program we would be adding several kilobytes of additional program. Some of it would be the DATA statements used to hold the information, some of it would be the RAM-consuming arrays into which the data would be put, and some of it would be the routines used to load in and sort out that data.

Instead of loading the data as part of the program we create our database first, then get rid of all the routines used to create that database, and save the whole block of data as one block of memory using *SAVE. Then all that is needed in the actual adventure are a few short routines to look at the appropriate section of memory and to read the required data. This is what has been done with the map data, the object data, and the dictionary. The dictionary is, however, a special case because, as another memory-saving device, most of the text used in the adventure has been compacted and coded. It is still held in DATA statements but these are roughly half the length of the text they represent. The dictionary held in our block of data is used to decode these compacted data statements as well as being used to check the commands input by the user. The result of this is that we save memory, we make the descriptions of the adventure virtually impossible for you to read as you type them in, thus creating a degree of surprise, we reduce the amount of typing a little (because the total of dictionary plus compacted text requires less typing than creating the whole text), we make a large potential vocabulary available for the user, and we enable very rapid decoding of input. (Can you imagine how long it would take to check a five-word sentence against a possible vocabulary of 140 words using the methods described in Chapter 7?)

The memory block consists of five main chunks arranged in this way:

1. All the data on the objects.
2. The map, held as a description of all the exits.
3. The legal verb that a player might type in.
4. The legal nouns that a player might type in.
5. The remainder of the dictionary needed for decoding descriptions.

Each of these blocks can be created in a number of ways. Those of us lucky enough to have a disassembler/monitor and/or a word processor will find the task a lot easier. A disassembler/monitor enables bytes to be put directly into memory and checked for accuracy. A word processor can make the compilation of dictionaries from a series of prose descriptions a much easier task. However, without these aids it is still a relatively simple task with custom-made routines. Essentially each routine is the same. It asks the user for the information to be encoded. Then it codes that information. Finally, it places that coded information into a secure place in memory. When all the information has been coded and stored in this way it is a simple task to save the complete block onto tape (or disk).

In the case of the three language blocks the main differences between them are simply in the way that the memory block is constructed. For the verbs and nouns it is necessary to have a regular arrangement so that any particular entry can be found very rapidly during the game. So the nouns are arranged one after the other in groups of eight bytes. These eight bytes will either be seven letters for the noun plus a <Return> character (coded as hexadecimal &OD and decimal 12) or if the noun is less than seven characters a number of <Return> characters to make up the difference. This is one of the inelegant aspects of this program because it wastes a few bytes of memory, but it is worth it for the rapid response in decoding input. Slow input processing is probably one of the major irritants in playing an adventure. The verbs are held in an identical fashion, but only seven bytes have been assigned to each verb because verbs are generally shorter than nouns. Two fudges have had to be made to make this system work, however. One is that a few nouns have been chopped short so that they fit the eight-byte format. These therefore require special tests in the input routine. In addition, two of the verbs in the verb list do not exist. There are spaces reserved in memory for a verb beginning with N and another with V but no such verbs are used. The reason for this is simply that access to the verb file is initially through the ASCII code of the left-most character of the word. The files are arranged alphabetically at the beginning of the file and therefore there has to be an 'n' and a 'v' in the list. The access routine is described in more detail below.

The third text block does not require rapid access so there is no need to arrange its contents in a regular format. Each word is given a byte for each character plus a <Return> character. When this part of the file is read in order to code text descriptions every string and substring is looked at one by one, which can take a minute or two for a long text, but time is unimportant at this stage. When the adventure is actually run and the coded strings are decoded the relevant dictionary entries are found immediately because the code holds the memory address of the relevant word so no search is needed.

The object data could have been held almost as efficiently as a byte array defined in the main body of the program. However, this would have required code for loading data into the array to be held and run during the adventure itself, which is too wasteful for our purposes. It is also rather easier to manipulate a block of memory than a byte array once you become accustomed to it because it is easier to find, load, and save the former.

The block holding the map data is slightly more complicated than the dictionary. It is held in what is known as a linked list (actually the same method is used as for the nouns and verbs). As there are 81 for locations or 'rooms' in the adventure there are 81 items in the first part of the list, each item describing the first exit in the room. As part of its description it points either to the next exit in the room, which will be held further down the block, or it has a marker which indicates that it is the last exit in that room, in which case no other items in the list will be examined.

The data structure for each of these is described below in the relevant section. A data structure is simply the format in which data are held. It is necessary to know and describe such structures because, as you know, all that a block of RAM actually holds is a series of numbers. Both programmers and programs need to know what those numbers mean and this will depend on the format used when they are read in.

Before looking at the data and the routines in detail let us briefly consider the design process behind this adventure. I will describe very rapidly the way that I went about constructing it.

Firstly, I ran the Ideas program, sometimes getting output on a printer to study later and sometimes just writing down ideas from the VDU. I did not take any notice of over 90 per cent of the ideas because they were either silly, trivial, or not what I wanted. Nor did I always write down the exact output. Instead I used the program to fire my own 'idea generator' and wrote down the modifications I thought up. This, together with some other ideas I had collected in a notebook, gave a list of about 40 interesting ideas.

Next I tried to group them together. I looked for ideas that were set in similar places or had related effects or used the same kind of object or fitted together in some other reasonably coherent way. Gradually by shuffling the ideas around I found I had six basic areas in the adventure. I drew up a crude map of six areas and inserted small boxes within each large area containing the ideas I still wanted to keep. As I did this fragments of plot began to suggest themselves and it was clear that certain linking objects or rooms had to be provided.

After drawing in these linking areas I could see what fitted together logically and what was somewhat isolated. These isolated areas were either crossed out or concentrated upon until some sensible connection suggested itself. For example, I had an underwater palace next to a cliff of ice. How could these two things fit together? Obviously the player has to get to the water's surface, so I gave the player a magnet which would be attracted by iron on the surface. As the player could not be directly under the cliff the iron had to be out at sea. I drew in an ice floe in which an iron bar was buried and made the floe float to the cliff. You will find that in the final game it does not happen in quite this way because I then decided to move the iron bar.

After all this I had something like a logical map. However, it looked a bit too logical so I moved one or two exits around, added one or two puzzles just to make things awkward, and created a few extra locations which were not actually 'in' the map in a logical way. At this point it became necessary to check that all the objects required to solve the puzzles were somewhere in the game and that it was always possible to get the required object to the necessary puzzle. This is an important part of the design process as it is all too easy to design a logic which fits together perfectly but which the player can never penetrate because A is needed to get B, B to get C, and C is needed to get A. Some people prefer to design the logical structure of their map long before they consider the actual plot or content in order to ensure that a solution actually exists.

At the same time it is necessary to check that there is not a possible pattern of simple solutions which solve the game without encountering half the map. Many adventures can be solved in more than one way and these are probably the best kind because they allow for creativity in the player. However, there should not be too many solutions and they should not be possible without having visited most of the adventure world.

At this point I began to write detailed descriptions of each of the rooms and objects. It is probably best to begin with some form of general sketch of each location, either in words or drawings, to get a general idea of the feel of the game and an idea of the likely vocabulary and probable length of the final descriptions. Take into account the fact that each puzzle and each solution, together with a fair number of errors, will require messages and may require extra code. As a rule you can assume that each location will want a description of about 20 words, that about half the locations will have hidden features which need a further description of about 12 words, and that you will need one error message for each verb and probably a help command for at least half the objects. If you have 50 locations, 20 verbs and 20 objects this means a total text of at least 2000 words. As the average length of a word is just over five characters this means an adventure of this size would need at least 10K for the text data alone. You can easily see why I have used text compaction in The Opal Lily. Every byte becomes vital.

When you have a reasonably complete description of each location, including all the various messages which might arise there, you will want to split that description into its logical parts. On the one hand you will have a description which will be printed every time that location is visited (plus perhaps a graphical equivalent). In addition there may be some additional messages which will only be revealed if certain conditions are met, such as visiting the location with the necessary object or pronouncing the correct password. A sensible way to deal with these is to give each location on the map a number and give the same number to each description, but using a subclassification for each of the additional messages at that location.

For example, location 12 might have three messages: 12a is always given, 12b is given when the orange is left there, and 12c is given when the player performs the dance of the seven veils.

Now we have a complete description of our adventure and its logical structure. Only at this point can we begin to think about coding it. With The Opal Lily I first drew up a block diagram of the major areas of the program, as shown in Fig. 9.1. I tried to make sure

00-59	Set up and Main Loop
60-199	Describe Current Location
200-999	Map
1000-2999	Process Input
3000-3999	End of Move and End of Game Routines
4000-6649	Verb Routines
6650-6999	Housekeeping Functions and Procedures
9000-9999	Room descriptions
10000-19999	Responses to actions
20000-20999	Object descriptions

Figure 9.1. Block diagram of the Opal Lily.

that I left enough space for all the likely sets of routines that would be required, but as usual I had to do a little tinkering with it as the design developed because I found I had miscalculated. Always err on the side of generosity when planning the amount of code you will need for a particular routine or block of program.

I also began to think about how to number the verbs and nouns so that I could easily keep track of them. To do this, of course, you need a list of all the verbs and nouns allowed in your game. Exactly how many items you have and how they are listed will depend on the method of storage and access you are using, However, remember that some nouns may be fixed to particular locations but most will wander around the adventure (being portable objects), that verbs usually are read first, and that your rooms should be numbered as far as possible in a logical way. I numbered my rooms according to the six major areas defined in the program. Having listed all the verbs I thought important I took one for each letter of the alphabet that I thought would be used most and arranged those in alphabetical order in the list. I did the same thing with the nouns. By doing this access will be speeded in interpreting commands and initial letters can be used for some commands if desired.

Now came the most tedious part of the design. I had decided what methods of storage I was going to use for the dictionary so I now had to decide exactly which words would be included in that dictionary. Of course, if you are not using this method of text compaction you will not need a dictionary, but you may still need to re-read your descriptions to see if there are too many words for the method you are using. The description of, say, 2000 words has to be reduced to a dictionary file consisting of all the words and word parts used in that description, with only one occurrence of each. For words like 'the' and 'a' it is easy to see that they occur more than once and so delete them from your file, but you might have a word like 'ebony' in the second location and also in the sixty-fifth one. If you do not notice that this word occurs twice it will be put into the dictionary twice and memory will be wasted.

Compiling a dictionary can be much easier by use of the 'search and replace' feature of a word processor. However, if you have handwritten or typed your own descriptions you will have to do the same thing by hand. Carry out the following algorithm on your description until you have looked at all the words in your file. It is a good idea to use alphabetical index cards on which to transcribe the dictionary in order to make organization and checking easier, but it is not essential.

1. START WITH THE FIRST WORD
2. WRITE INTO THE DICTIONARY THE NEXT UNDELETED WORD IN THE FILE
3. GO THROUGH THE FILE AND DELETE EVERY OCCURRENCE OF THAT WORD
4. IF ALL THE WORDS IN YOUR FILE HAVE BEEN DELETED THEN YOU HAVE FINISHED IT; OTHERWISE GO TO THE NEXT UNDELETED WORD IN THE FILE AND GO TO STAGE 2.

Obviously it is a good idea to have a separate copy of your description because you will have to scribble all over the work file. It is best to ensure that the two files are identical by getting either a printout or a photocopy. If you intend to use the dictionary to check input in the manner of The Opal Lily you can use your dictionary to mark the legal input verbs and nouns because these will have to be identified separately in your dictionary for processing in different ways.

Now we are ready to code.

9.2 Objects

A crucial part of any adventure built around a puzzle is the number and nature of objects that can be found and what can be done with them. Some form of record needs to be kept of these objects so that the program knows what the player has found and what has been done with them. Many adventures use a very simple system of flags. A flag is simply a numeric variable which is altered to mark the status of some aspect of a program. It may be used to 'flag' the fact that a particular condition is now TRUE or FALSE or it may record more complex conditions. Each object is given a flag and as the status of the object changes so the flag changes. Every time the player attempts to perform an action the flag is consulted. If it is set to the correct number the action is permitted; otherwise the action is now allowed.

Moving the available objects around the world is the crucial aspect of the puzzle adventure game from the player's point of view. The player should be allowed to take any object anywhere, provided that the necessary puzzles can be solved. Thus at any stage in the game we need to know where any object is and which objects the player is carrying. The character must be able to pick up objects (The Opal Lily uses the common verb TAKE for this) and to leave them behind (The Opal Lily uses DROP). We might also allow an INVentory command so that the player can be reminded of what he or she is carrying.

The Opal Lily takes care of these problems by using a three-byte data structure for each object. As there are 87 nouns (hence 87 potential objects, though some of the nouns cannot usefully be used by the player) this means a block of 261 bytes. Byte 1 holds the current room number of the object, a number between 1 and 81. It can also be set to various other values: 0 means it no longer exists (perhaps the player has destroyed it), 99 means the player is carrying it. This is thus a flag with 83 useful states, but as it could be set to any value up to 255 we may find other uses for it as well.

Bytes 2 and 3 are used to mark various aspects of the status and nature of all the objects. One of the non-intelligent features of a number of adventures is that they allow commands like BURN STICKS but then count BURN DOOR or BURN LEAVES as meaningless. This is because burning sticks is an important feature of the particular adventure, but burning wood or doors is irrelevant to the main puzzle. Not only is this frustrating to the player but it is also rather unrealistic and irritating. It would be much better if our game allowed each verb to be used with each legal noun in a logical way which reflects the real world, but only some of those actions will be useful in solving the adventure. This adds one further level of puzzlement to the game. The player knows something must be burnt to frighten away the bear, but does not know what will do the trick and in experimenting may destroy the magic wand which is needed to solve a later puzzle.

If we were to code every possible relation of verb to noun with a routine for each combination we would find ourselves with an impossible task. Suppose we had 20 verbs and 50 nouns. Then there are 1000 possible combinations, each of which would require a routine! It is obviously simpler to have one single "You can't do that" routine for the majority of these combinations, only allowing puzzle-solving relations, but this is dull programming. What is really needed is some development of semantic coding, the kind of thing which is the basis of much AI research, perhaps similar to that discussed for the Ideas program in Chapter 8.

The Opal Lily does not try to anticipate every possible relationship, but it does allow a number of actions which would be possible in the real world but are not very useful in the game. It does this by coding a number of semantic categories into the two status bytes for each object and then using these categories to decode the appropriate verb routines. For example, suppose the player wanted to fill the chest. The purpose of the chest in the program is simply to hide one of the treasures so that once it has been opened it serves no further purpose. However, a chest is a container so if the player wants to fill it that should be allowed. So we record in the status bytes whether every object is a container or not and if so whether it is full or not. This information is actually only important for the shovel/spade, but the player does not know that.

Similarly, these bytes hold information about whether an object is alive, whether it can move, whether it can be broken, whether it is broken, whether it can be eaten or drunk, whether it has been eaten or drunk, if it can be worn, if it is being worn, if it is concealing another object, if it is concealed by another object, and if it can be or has been burned. In addition, there is a general flag which has four possible states and holds special information for certain objects. This means that in 174 bytes we hold enough information to interpret roughly 1200 relationships of noun to verb, because 14 verbs use this information and there are 87 nouns. Of course, in the majority of cases the result is still negative. If the player tries to MOVE FARM or BURN WATER or EAT RAFT the attempt will be unsuccessful. Also, there are still many actions which ought to be allowed and are not, such as EAT GRASS, which is possible but foolish, or BURN CHIEF, which is not allowed because it might mean 'set fire to the chief' or it might mean 'hurt him' and the program has no way of knowing which. Nevertheless, the idea is clearly a good one and developments along these lines are what we should expect from commercial games. After all, if this amount of semantic information can be stored in such a small space one might be able to write adventures which work purely by interpreting general semantic relations rather than by trying to envisage all the separate individual cases.

How is all this information held in two bytes? You will remember that a byte is made up of eight bits and that each bit can be in one or two states, either on or off, represented as either 1 or 0. Each bit in a byte can therefore be regarded as a separate flag. So in two bytes we could hold 16 flags of the status of an object. In the first byte one could, for example, represent 'breakable' if set to 1 and 'unbreakable' if set to 0. However, we can get a little more information in if we group the bits into pairs. Thus the leftmost pair of bits in the first byte holds the following information in The Opal Lily:

00	Not edible or drinkable
01	Drinkable but not consumed
10	Consumed
11	Edible but unconsumed

These two bits can be used to answer four questions:

1. Is it edible?
2. Is it drinkable?
3. Has it been eaten?
4. Has it been drunk?

These are all 'Yes/No' questions so each could be represented by one bit of information, but then we would want four bits, not two, which doubles the amount of memory needed. The complete structure of the two status bytes is:

Status byte 1	Bits
Consumable?	7/6
Container?	5/4
Wearable?	3/2
Hiding/hidden?	1/0

Status byte 2	Bits
Breakable?	7/6
Mobile/alive	5/4
Burnable?	3/2
Special?	1/0

Each time a verb is called it will be decoded with reference to the appropriate pair of bits. For example, suppose the player types "BREAK ROCK WITH BOTTLE". The routine for BREAK will look at bits 7 and 6 of status byte 2 for both rock and bottle. It will find that rock is set to 00 so the rock cannot be broken. However, the bits for bottle will be set to 01 so it can be broken. Probably it will be, so the two bits are reset to 10, meaning 'the object has been broken'. Now it may be that a broken bottle will be useful elsewhere in the adventure, so this resetting will become important. On the other hand, the bottle may now be useless, in which case the 'possession' byte will be set to 0 because the object has effectively been destroyed.

As an indication of what the special pair of bits might mean, let us take the example of the logs and the vines. Initially the special bits for both these are set to 00. They will be set to 01 when the player has found them. When the vines are tied to the cliff to enable climbing of it they are set to 10, signifying 'tied'. They then have to be untied at the top of the cliff, i.e., set to 01 again, in order to be carried to the river, where they are tied again, this time to the logs, to make the raft, which sets the vines to 11 and the logs to 11. When the raft breaks both are set to 00 and the possession byte is also set to 0 for 'destroyed'. Of course, if meanwhile the player types "BURN VINES" or "BREAK LOGS" it may be found that the items are destroyed prematurely and a raft can never be made.

The general routines which make use of the object file are those which describe the objects to be found in a location, which give the player an inventory of current possessions and which allow objects to be taken and dropped. Naturally there is a command for each of these actions. In the first case, after describing the location the routine will look through the whole object file for objects whose first byte (the possession byte) is the same number as that of the current room. When it finds one it checks the 'Hidden?' bit of status byte 1 and if the object is revealed it will print its name, giving a list of all such objects. If the player LOOKs at some of these locations the hidden bit, if set, may be reset thus revealing an object.

For an INVentory the same routine as the description routine looks for bytes which equal 99. DROP changes the possession byte from the player's number to the room number and TAKE does the reverse. However, DROP also has a slight quirk to make it a little more interesting. A variable records the number of times the player uses the verb DROP. The higher the variable the more likely the player is to break something which is dropped, provided it is breakable. The rationale behind this is that repeatedly dropping things implies clumsiness and hence a likelihood of breaking things, and in game terms the more a player is undecided about what strategy to use, what to take, and what to leave, the greater the risk that something will go wrong. The idea is thus to plan ahead in what you carry and only take what you intend to use.

9.3 The map

The method for storing the map of The Opal Lily is not any of those described in the previous chapter. Instead a 'linked list' has been used. A linked list is a list of items held in memory in which each item in the list points to the next relevant item. The reason for using this method is that it was necessary to store several pieces of information about each exit, in a similar way to storing information about each object. The program needs to know for any given location how many exits there are, which direction they go, what locations they lead to, and whether they are currently available to the player. The drawbacks with the methods given in the previous chapter are either that they use a great deal of memory, or they could not hold all this information, or they cannot adequately describe a map in which rooms are not immediately next to each other.

For every room in the adventure there are six possible exits—north, south, east, west, up, and down. However, no room has all these exits and most have less than four. Thus if we were simply to set up a two-dimensional array with one dimension the number of rooms and the other the number of exits, roughly half the slots in the array would be unused. At four bytes per slot for an integer array this is a loss of (81 * 6 * 4)/2 bytes or nearly 1K! What is needed is a structure that holds the necessary information on each exit, but does not even hold empty spaces for the missing exits. As there is no way of predicting how many exits there will be from a particular room any such list will have an irregular structure. It is therefore necessary for the entry for each exit in a room to point to the entry for the next exit. This is how the records in the list are linked. For example, the first exit in room 1 goes south. The entry in the list says that it can be used by the player at once and it goes to room 2 and that the next exit can be found 81 entries further down the list.

So the data structure for each exit consists of a three-byte entry. The first byte holds the direction of that exit in bits 0 to 5, whether the exit is hidden or not in bit 6, and whether it is the last exit for this room in bit 7. The second byte holds the number of the room the exit leads to and the third holds the offset, if any, to the next relevant record in the list. It does not hold the actual number of bytes separating the two entries but the number that is to be multiplied by three to give the required address.

On entering a room each time the description routine looks first at the item in the list with the room number, e.g., for room 27 it looks at the twenty-seventh item in the list. If that exit is hidden it looks at byte three for item 27 and if that is zero it will print "No exits". However, if the room is not hidden it decodes the relevant bits of byte 1 and prints the direction that it leads. If byte 3 is not zero it will then calculate the address of the next relevant entry and repeat the routine for that item. It continues in this way until it finds an entry whose third byte is zero, when it will finish the routine.

The movement routine performs in exactly the same way, except that instead of printing the chosen exit it will print "You can't go that way" if there is no exit or if the exit is hidden, but will look at byte 2 of the entry and change the location variable R% to that byte if there is an available exit.

In order to put the map into memory in the first place it is necessary to run a routine which does the following. First it must ask the programmer for the total number of rooms in the adventure and then for the DATA on each room in turn. It must code the information into the relevant bytes, placing the first three-byte code per room at the location in memory represented by the number of the room, then working out the address of the next free byte after the total number of first-byte entries, placing that exit there, and putting the address onto the third byte of the previous entry. When all the rooms have been worked out and stored in RAM the block must be saved in the same way as other blocks of DATA using *SAVE.

However, to make things easier for you the object and map data for The Opal Lily is contained in the programs listed in Figs. 9.2 and 9.3. It is a very simple program which just reads all the map information held in the DATA statements into memory starting at location &4DA7. Then it *SAVEs the whole block as a file called "MAP".

```
  9  REM LOADS OBJECT DATA TO &4C98
 10 HIMEM=&4C97
 20 H=HIMEM+1
 40 FORI=0TO260
 50   READ X
 60   ?(I+H)=X
 80   NEXT
 89  REM LOADS MAP  DATA TO &4DA7
 90 H=&4DA7
100 FORI=0TO587
110   READ X
120   ?(I+H)=X
130   NEXT
140 *SAVE "MAP" 4C98 4FFF
190 END
```

Figure 9.2 Program to load object and map data.

```
99 REM OBJECT DATA
400 DATA 12,128,32
401 DATA 11,33,96
402 DATA 34,33,72
403 DATA 0,0,0
404 DATA 0,0,0
405 DATA 22,0,104
406 DATA 36,0,97
407 DATA 10,128,41
408 DATA 0,128,104
409 DATA 17,49,97
410 DATA 18,0,104
411 DATA 9,136,104
412 DATA 0,64,41
413 DATA 0,0,0
414 DATA 32,8,104
415 DATA 30,128,120
416 DATA 0,1,16
417 DATA 0,16,105
418 DATA 0,0,0
419 DATA 34,0,1
420 DATA 0,0,0
421 DATA 6,136,104
422 DATA 0,0,0
423 DATA 0,1,88
424 DATA 3,57,105
425 DATA 71,33,104
426 DATA 3,8,104
427 DATA 0,0,104
428 DATA 0,0,0
429 DATA 0,0,32
430 DATA 0,1,112
431 DATA 0,0,0
432 DATA 45,33,72
433 DATA 0,1,0
434 DATA 0,0,32
435 DATA 0,1,112
436 DATA 0,1,72
437 DATA 57,1,104
438 DATA 0,0,0
439 DATA 39,0,104
440 DATA 0,64,112
441 DATA 0,0,32
442 DATA 38,136,112
443 DATA 0,136,40
444 DATA 0,0,0
445 DATA 0,128,120
446 DATA 0,0,0
447 DATA 17,0,97
448 DATA 0,8,105
449 DATA 5,16,104
450 DATA 0,56,105
451 DATA 63,0,32
452 DATA 0,0,0
453 DATA 24,8,32
454 DATA 16,0,120
455 DATA 69,0,120
456 DATA 83,8,105
457 DATA 47,8,104
458 DATA 0,0,96
459 DATA 0,0,104
460 DATA 16,128,40
461 DATA 0,136,120
462 DATA 20,0,104
463 DATA 0,128,32
464 DATA 61,16,96
465 DATA 19,0,104
466 DATA 10,136,104
467 DATA 0,0,8
468 DATA 0,0,0
469 DATA 0,0,112
470 DATA 0,0,16
471 DATA 0,1,16
472 DATA 2,64,41
473 DATA 78,0,32
474 DATA 0,0,88
475 DATA 0,64,0
476 DATA 0,128,0
477 DATA 0,1,72
478 DATA 0,0,40
479 DATA 0,0,0
480 DATA 0,0,32
481 DATA 0,0,0
482 DATA 2,129,8
483 DATA 0,0,8
484 DATA 0,0,0
485 DATA 0,64,0
486 DATA 0,0,0
```

Figure 9.3 Object and map data for Fig. 9.2 (*continues*)

```
499 REM MAP DATA
501 DATA 32,8,82,32,7,84,129,7,0
502 DATA 66,6,84,8,7,85,136,8,0
503 DATA 8,4,85,160,12,0,32,13,84
504 DATA 8,10,85,32,22,86,32,14,88
505 DATA 132,18,88,136,16,0,32,21,89
506 DATA 8,22,89,8,19,0,32,25,89
507 DATA 96,23,91,96,58,91,8,23,92
508 DATA 32,55,95,16,27,96,32,26,96
509 DATA 72,26,96,2,28,96,32,29,97
510 DATA 32,30,99,136,31,0,0,27,98
511 DATA 160,33,0,96,37,98,16,30,98
512 DATA 96,39,98,132,37,0,8,40,99
513 DATA 2,42,99,194,23,0,65,40,98
514 DATA 1,42,98,32,46,99,160,44,0
515 DATA 80,44,99,136,46,0,8,50,100
516 DATA 16,48,101,32,53,101,8,53,101
517 DATA 96,51,102,32,50,103,96,53,104
518 DATA 72,56,105,32,59,106,8,58,107
519 DATA 32,60,107,96,64,108,80,58,110
520 DATA 66,63,111,160,61,0,2,65,111
521 DATA 144,59,0,65,63,110,1,65,111
522 DATA 96,68,111,96,48,113,8,79,114
523 DATA 32,69,115,32,70,115,8,71,116
524 DATA 96,67,116,32,75,117,96,76,118
525 DATA 132,75,0,32,78,118,16,77,118
526 DATA 72,80,118,96,81,119,144,80,0
527 DATA 132,5,0,132,3,0,8,2,1
528 DATA 132,12,0,8,5,1,16,3,1
529 DATA 132,10,0,8,1,1,132,4,0
530 DATA 16,5,1,132,8,0,16,4,1
531 DATA 132,9,0,132,13,0,8,3,1
532 DATA 208,11,0,16,12,1,194,14,0
533 DATA 65,13,1,16,15,1,132,19,0
534 DATA 132,16,0,8,15,1,16,17,0
535 DATA 132,18,0,144,16,0,16,14,1
536 DATA 132,20,0,68,26,1,16,18,0
537 DATA 196,26,0,8,14,1,16,19,1
538 DATA 196,24,0,16,22,1,132,25,0
539 DATA 16,22,1,132,25,0,8,24,1
540 DATA 144,21,0,132,28,0,208,33,0
541 DATA 1,29,0,16,30,1,132,35,0
542 DATA 16,31,1,132,36,1,8,33,0
543 DATA 196,32,0,16,34,1,132,30,0
544 DATA 200,29,0,160,37,0,72,38,1
545 DATA 80,35,1,68,36,0,144,37,0
546 DATA 132,39,0,130,43,0,2,44,1
547 DATA 136,52,0,16,43,1,4,45,0
548 DATA 68,47,1,96,48,1,136,51,0
549 DATA 16,46,1,132,49,0,196,36,0
550 DATA 132,48,0,16,52,1,132,46,0
551 DATA 68,43,1,136,54,0,4,51,1
552 DATA 16,54,0,68,52,1,144,55,0
553 DATA 16,25,1,196,54,0,8,57,1
```

```
554 DATA 144,55,0,132,56,0,16,23,1
555 DATA 132,57,0,72,60,1,80,56,1
556 DATA 68,61,0,68,59,1,96,65,0
557 DATA 8,59,1,144,62,0,1,61,0
558 DATA 16,60,1,136,66,0,130,67,0
559 DATA 72,71,1,80,73,1,132,66,0
560 DATA 8,69,1,144,67,0,16,70,1
561 DATA 132,68,0,136,71,0,16,72,1
562 DATA 132,67,0,144,73,0,4,72,1
563 DATA 200,74,0,4,73,1,193,8,0
564 DATA 72,77,1,144,74,0,132,75,0
565 DATA 132,79,0,80,78,1,132,69,0
566 DATA 132,79,0
```

Figure 9.3 Object and map data for Fig. 9.2

The program for loading map DATA can be used with the data for any adventure using the same format. However, you need the DATA statements in the first place so the program listed in Fig. 9.4 will turn information typed in when it is run into a pseudo DATA statement. This is printed out on the screen at the end of execution. If you then use the <Copy> key to copy it that will be added to the program. Keep running the program and adding the new DATA statements until all the exits have been coded. Make sure that you do one exit for each room before doing any others. All exits which are the only exits in a room will be given a 0 as the third byte but the others will be given a dummy "*". When you have completed all the DATA you must work through it and replace all the asterisks with the number of lines separating two subsequent DATA statements for a room from each other.

```
 9 REM ROUTINE TO CODE EXIT DATA FOR ROOMS

10 CLS

20 PRINT'''

30 dat%=0

40 PRINT"WHAT DIRECTION?"

50 D$=GET$

60 x= INSTR("NSEWUD",D$)

70 IF x=0 THEN 40

80 dat%=dat%+2^(6-x)
```

Figure 9.4. Exit data coder *(continues)*

```
 90 PRINT "Is exit available?"
100 yn$=GET$:IF INSTR("YNyn",yn$)=0 THEN 90
110 IF yn$="N" OR yn$="n" THEN dat%=dat%+64
120 PRINT "is this the last exit in the list?"
130 yn$=GET$:IF INSTR("YNyn",yn$)=0THEN 120
140 IF yn$="y" OR yn$="Y" THEN dat%=dat%+128
150 PRINT"what room does it lead to?"
160 A%=A%+10
170 INPUT room
180 PRINT A%;" DATA ";dat%;",";room;",";
190 IF dat%<128 THEN PRINT"**" ELSE PRINT"0"
200 PRINT''
210 PRINT "Now <Copy> this DATA statement"
220 PRINT "To add it to the program"
```

Figure 9.4. Exit data coder.

To make this clearer, suppose you have finished running the program for your adventure and room 1 has two exits. You will find that the third element of its first DATA statement has an *. If you now look for the DATA statement coding the next exit in room 1 and subtract the lower from the higher you will get the offset which is to replace that asterisk. Suppose the first exit is in statement 1000 and the second in statement 1083. Then the asterisk must be replaced by 83. You will know which statement to look for because it is the first one after the last room in the adventure, i.e., it will be the eighty-second DATA statement for The Opal Lily because there are 81 rooms.

Similarly, repeat the operation for each room and each exit in each room, looking for all the exits from each room until the statement ending in 0 is found. One way to make each statement easy to find is to use a sensible numbering scheme for the DATA statements. They can always be renumbered afterwards if necessary. A possible scheme would be that each statement will end with the room number and begin with the exit number. So exit 1 for room 43 would be line number 143; exit 2 would be 243; exit 3 would be 343. If you have more than 99 rooms then use 1000 as your baseline number.

9.4 Encoding and decoding the text

The descriptive text for The Opal Lily is held in DATA statements from line 9000 onwards. You have to type these in exactly as they are, even though they look like nonsense. This is because each line actually consists of a series of numbers represented by the ASCII code of the characters in the DATA strings. The numbers are in pairs and represent a number system in base 64. What this means is that, whereas, the decimal system consists of 10 digits from 0 to 9, the hexadecimal system consists of 176 (0,1,2,3,4,5,6, 7,8,9,A,B,C,D,E, and F) so a base 64 system has 64 codes. The right-hand letter of each pair thus represents a digit between 0 and 63 and the left-hand letter represents a number between (0*64) and (63*64). The maximum range of a two-digit code using base 64 numbers is thus ((63*64) + 63), i.e., 4095.

These 4095 numbers are the possible locations that can be addressed by this system. Therefore we can store up to 4095 pieces of information (each one, one byte long) and recall it by using the unique two-character code for its address. However, as the address code is two bytes this is rather a foolish way to store items of data which are one byte long. For every byte of data we need two bytes of address data, so to store 100 characters would cost 300 bytes! However, suppose the data being stored was longer than three bytes. It would still seem more costly on memory to hold three-byte items in memory plus two-byte addresses than it would be to hold the data in data statements or arrays. This is true, provided each item would only have been stored once in the DATA statements.

However, suppose our data was "THE GOBLIN IN THE DUNGEON TAKES THE PINEAPPLE". Including spaces this string is 45 characters long, hence 45 bytes (plus the cost of the DATA line itself). If each of the words was held at one of our 4095 addresses and the DATA statement just held the coded addresses then we could write a procedure which recalled each word in turn by looking at each address in the DATA string in turn, PRINTed it, and then PRINTed a space. This would mean 16 bytes to hold the addresses (eight words * two-byte code) and 32 bytes of memory for the characters in the string. It is 32 because we can forget the spaces (7 bytes) and the word THE is used three times but we only need to store it once and use the same address code three times. 16 + 32 = 48, which is still more than the original 46, but not much.

Now suppose that another description in the program is "THE PINEAPPLE IS IN THE DUNGEON". How many bytes to hold this? There are 31 bytes in the literal string. If we used the same codes as the previous description we would need six words * two bytes = a twelve-byte string. All of the words except 'IS' are in the string previously encoded, so we need two more bytes to put this into memory, giving a total of 18 bytes used thus saving 13. If we add the two strings together then it would take 46 + 31 = 77 bytes to hold these strings as DATA statements but only 48 + 18 = 66 using the memory block and address code system.

It is not, as you might expect, quite as simple as this. In the first place we must add to each word stored one more byte for the <Return> character. This is because the $ operator available on the BBC and Electron allows us to read any string starting at a specific address and ending with the <Return> character. If we did not have this facility we could still retrieve strings from memory but we would have to store the strings at regular intervals and retrieve them one byte at a time starting at a given position. So our total memory cost would be 66 bytes plus 7 return characters = 73. Therefore all we have saved is 4 bytes!

This might not appear large, but if you have two sentences for each room and one hundred rooms in your adventure then you will have saved nearly 1K in all. The savings are even much greater than this. If you think about it the saving can be expressed in a rough and ready way as a formula. If the length of the word to be encoded plus a space = L and the number of times it is used in the total adventure is N, then the total saving is:

$$\text{Saving} = ((L*N) - ((2*N) + L)) \text{ bytes}$$

Thus if the word is one character long and used 10 times we have:

$$\begin{aligned}\text{Saving} &= ((2*10) - ((2*10) + 2)) \\ &= 20 - 22 \\ &= -2\end{aligned}$$

This is a negative result and hence wasteful. However if we have a six-letter word used five times then:

$$\begin{aligned}\text{Saving} &= ((7*5) - ((2*5) + 7)) \\ &= 35 - 17 \\ &= 18\end{aligned}$$

which is a saving of over 50 per cent.

In general, therefore, the longer a word is and the more it is used in an adventure the greater will be the saving using this method. Remember, however, that you are limited in total memory store to 4095 bytes (a little less than 4K) so there is an absolute limit on the number of words in your dictionary, which is another argument for repeating words as often as possible. The potential dictionary could be expanded by using a number code with a base larger than 64. A base of 100 would give us 9999 addresses or nearly 10K for our dictionary.

Other ways to make the system more efficient are to create two dictionaries and to use word parts rather than entire words. The first method has not been used here but one can use one-byte codes to represent the very short or very frequent words. If the characters representing single-byte codes are a different set from those encoding the two-byte addresses the decoding routine can check for them and branch accordingly. For example, if we were using a code system in base 26 then lower case letters could be interpreted as one-byte codes accessing a one-byte dictionary and upper case letters could be read in pairs as accessing a separate two-byte dictionary. Obviously the one-byte address system only allows a small dictionary, limited to the base of number chosen, but it means that the inefficiencies of using three bytes to encode a two-byte word can be avoided.

The second alternative has been implemented in The Opal Lily. Instead of handling just words it deals in what linguists call morphemes, i.e., pieces of words that have their own meaning and can be combined with different words. Thus the simple coding system would hold QUICK, SLOW, QUICKLY, and SLOWLY at four different addresses. It would obviously use less memory to have only three entries for SLOW, QUICK, and LY. There are a large number of bits of words that recur like this and some words can be thought of as ending with such bits even if they could not normally be detached. For example, taking the LY from FLY does not give a meaningful word but it would be one way of coding it.

The problem is that word endings have to be printed immediately after the word beginnings but our decoding routine puts a space after each dictionary entry it decodes. If left to itself it would thus PRINT "SLOW LY" rather than "SLOWLY". We thus have to build the backspace character into the system. This is ASCII character 127, one of those invisible characters that can be used in VDU statements. If we add to each word ending a '(' character and regard this as meaning 'backspace' then on decoding the bracket can be swapped for CH$(127) before printing. This is done in line 1595 of the program. It means using one more byte for each word part entered in the dictionary, but if that means not having to enter the same word beginning twice this will mean an overall saving. The SLOW/QUICK example above would use 26 bytes to store information that word splitting can encode in 15.

We can now look at the actual encoding and decoding routines used. First the dictionary must be placed in memory. Run the program in Fig. 9.5 and it will fill the memory from &5000 onwards with a dictionary defined in the DATA statements, organized in the way described above. Then it will ask for a blank tape and *SAVE the whole file onto tape as a file called “DICT”. Next time you need the dictionary you need only *LOAD the DICT file. You can easily change the DATA statements to create a different dictionary for use in another adventure which uses the same coding system.

```
  5 REM Reads Dictionary into Memory at &5000
  9 REM First the verbs
 10 P=&5000
 20 FOR i= 0 TO 51
 30 READ verb$
 40 READ m
 50 $(P+(i*7))=verb$
 60 ?(P+6+(i*7))=m
 70 NEXT
 99 REM Next the Nouns
100 RESTORE 600
110 Q=&5170
120 FOR i= 0 TO 86
130 READ noun$
140 READ m
150 $(Q+(i*8))=noun$
160 ?(Q+7+(i*8))=m
170 NEXT
199 REM Finally the rest of the words
200 RESTORE 700
210 R=&5427
220 REPEAT
230 READ text$
240 $R=text$
250 R=R+LEN(text$)
260 UNTIL text$=""
299 REM Now save the data
300 PRINT "Place a cassette in the recorder"
310 PRINT "Then press any key"
320 g=GET
330 *SAVE "DICT" 5000 5FFF
340 END
499 REM ***   DATA   ***
```

Figure 9.5 Dictionary loader (*continues*)

```
500 DATA "ASK",  0
501 DATA "BREAK", 22
502 DATA "CLIMB", 24
503 DATA "DROP", 24
504 DATA "ENTER", 26
505 DATA "FILL", 27
506 DATA "GO", 31
507 DATA "HOLD",  0
508 DATA "INVEN",  0
509 DATA "JUMP",  0
510 DATA "KILL",  0
511 DATA "LOOK", 27
512 DATA "MOVE", 29
513 DATA "N",  0
514 DATA "OPEN",  0
515 DATA "PLANT",  0
516 DATA "QUIT",  0
517 DATA "REAP",  0
518 DATA "SHUT", 26
519 DATA "TAKE", 30
520 DATA "UNTIE",  0
521 DATA "V",  0
522 DATA "WEAR", 29
523 DATA "BRIBE",  1
524 DATA "BUILD",  1
525 DATA "BURN",  0
526 DATA "CUT",  0
527 DATA "DRINK",  1
528 DATA "DIG",  1
529 DATA "DOUSE",  0
530 DATA "EMPTY",  1
531 DATA "EAT",  0
532 DATA "FIGHT",  1
533 DATA "FILE",  1
534 DATA "FLY",  1
535 DATA "FAN",  1
536 DATA "FLOAT",  0
537 DATA "GIVE",  0
538 DATA "LAUNC",  1
539 DATA "LIFT",  1
540 DATA "LURE",  0
541 DATA "MAKE",  1
542 DATA "MELT",  1
543 DATA "MOUNT",  0
544 DATA "SWIM",  1
545 DATA "SMASH",  1
546 DATA "SMEAR",  1
547 DATA "SKATE",  1
548 DATA "SHADE",  0
549 DATA "THROW",  1
550 DATA "TIE",  0
551 DATA "WAVE",  0
```

Figure 9.5 (*continues*)

```
600 DATA "APPLE",23
601 DATA "BOTTLE",24
602 DATA "CHEST",28
603 DATA "DOWN",31
604 DATA "EAST",32
605 DATA "FAN",33
606 DATA "GEM",75
607 DATA "HONEY",39
608 DATA "ICE",0
609 DATA "JAR",0
610 DATA "KEY",37
611 DATA "LEAVE",37
612 DATA "MOON",39
613 DATA "NORTH",41
614 DATA "OPAL",0
615 DATA "PILOT",40
616 DATA "QUEEN",0
617 DATA "RAFT",40
618 DATA "SOUTH",42
619 DATA "TEETH",48
620 DATA "UP",48
621 DATA "VINE",48
622 DATA "WEST",48
623 DATA "APE",1
624 DATA "AQUA",0
625 DATA "BAG",1
626 DATA "BEAD",1
627 DATA "BONE",1
628 DATA "BATTLE",1
629 DATA "BREATH",49
630 DATA "CHIEF",1
631 DATA "CLIFF",1
632 DATA "COFFIN",1
633 DATA "CORPSE",0
634 DATA "DEEP",1
635 DATA "DRAGON",0
636 DATA "EBONY",1
637 DATA "EXPLOD",0
638 DATA "FARM",1
639 DATA "FILE",1
640 DATA "FISH",1
641 DATA "FLAME",1
642 DATA "FLAT",1
643 DATA "FLOWER",1
644 DATA "FLUE",1
645 DATA "FLYING",34
646 DATA "HOLE",37
647 DATA "KNIFE",0
648 DATA "LASSOO",1
649 DATA "LOG",1
650 DATA "LUNG",0
651 DATA "MAGNET",1
652 DATA "MOUTH",1
653 DATA "MUD",0
```

```
654 DATA "NATIVE",0
655 DATA "PEGASU",1
656 DATA "PLUME",28
657 DATA "ROD",1
658 DATA "ROCK",1
659 DATA "RUBY",23
660 DATA "SEED",1
661 DATA "SKATE",1
662 DATA "SCYTHE",1
663 DATA "SNOW",1
664 DATA "SPADE",1
665 DATA "STICK",1
666 DATA "SUN",0
667 DATA "TREE",0
668 DATA "UNDER",0
669 DATA "VILLAG",0
670 DATA "WASP",1
671 DATA "WHIRL",1
672 DATA "WHISKY",1
673 DATA "WHITE",1
674 DATA "WITCH",1
675 DATA "WATER",1
676 DATA "WEED",1
677 DATA "WALL",9
678 DATA "BROKEN",0
679 DATA "FIELD",1
680 DATA "FIRE",0
681 DATA "GATE",1
682 DATA "REED",3
683 DATA "HUT",0
684 DATA "PIG",0
685 DATA "RIVER",0
686 DATA "WIND",79

700 DATA "OF","IN","AT","ON","BY"

701 DATA "WITH","TO","THROUGH","OVER","ABOVE"

702 DATA "OUT","AND","A","AN","IS"

703 DATA "AS","ARE","THE","THAT","HAS"

704 DATA "THERE","I","YOU","HE","THEY"

705 DATA "IT","(Y","(THRONE","(D","(EST"

706 DATA "(ED","(LONG","(S","(CONTROLLAB","(PLE"

707 DATA "(DY","(LY","(ING","(ES","(LESS"

708 DATA "(EN","(WAY","(TO","(ERY","(LOW"

709 DATA "(BOW","(CIRCLE","(ILY","AL","EN"

710 DATA "UN","RAIN","WAR","CEIL","SEE"
```

```
711 DATA "VAST","EXPANSE","ROOF","JUNGLE","FRONT"
712 DATA "LARGE","SWEET","SMELL","LIKE","HUGE"
713 DATA "SOME","PATCH","GAUDY","MANDRAKE","VIBRAT"
714 DATA "OMINOUS","FLOUNDER","FOUL","SWAMP","GROWN"
715 DATA "PATH","TOP","HIGH","BRANCH","LEAF"
716 DATA "SCREECH","GIANT","NEST","RISE","AHEAD"
717 DATA "SURROUND","BUSH","CANOPI","SIDE","CAVE"
718 DATA "WHERE","OLD","STILL","LIE","TALL"
719 DATA "WOOD","FENCE","CONFRONT","EITHER","CARVED"
720 DATA "FIGURE","MAN","MASK","WAV","CLOUD"
721 DATA "DUST","STOCKADE","SEVERAL","DO","VARIOUS"
722 DATA "MENIAL","JOB","ALL","DRIED","CLAY"
723 DATA "MAK","HEADRESS","GROUND","LITTER","CORPSE"
724 DATA "OTHER","ARMOUR","SHELL","CLOTH","SEA"
725 DATA "CLEAR","HERE","FLOW","STAIN","GOLD"
726 DATA "WARM","DOCTOR","TEM","MADE","BLOCK"
727 DATA "WAY","POOL","BUBBL","RUNN","STEAM"
728 DATA "HOT","SWIRL","AROUND","R","CRASH"
729 DATA "AGAINST","PROW","WRECK","SHIP","LOOM"
730 DATA "STRAND","BARREN","FIND","LEAD","SHARK"
731 DATA "TOOTH","GARDEN","FERN","GROW","REGULAR"
732 DATA "PLOT","SAND","EACH","MARK","ROW"
733 DATA "TURN","TWIST","GRIN","BASK","SUBMERGED"
734 DATA "PORTION","ROTT","SKELETON","DRESS","PROPP"
735 DATA "GLADE","BLUE","FOUNTAIN","SPOUT","SILVER"
736 DATA "CORAL","MAZE","FOREST","TURQUOISE","PALACE"
737 DATA "RUIN","DECAY","CAUGHT","VIOLENT","FORC"
738 DATA "(WARD","DESERT","ISLAND","FLOE","CRACK"
```
Figure 9.5 Dictionary loader (*continues*)

```
739 DATA "BASE","STEEP","IDE","MOUNTAIN","FROZEN"
740 DATA "LAKE","THIS","CHAMBER","GREAT","CRAG"
741 DATA "TRANSLUCENT","HANG","(RN","STAND","(MAN"
742 DATA "CHILL","TONE","SAY","BLACK","STONE"
743 DATA " "","SLAB","MALACHITE","SIT","SCAR"
744 DATA "(RED","SHE","SMILE","WICKED","FLEX"
745 DATA "LONG","JAGGED","CLAW","STUMBLE","PIT"
746 DATA "RO","BE","(YOND","SIGHT","CENTRE"
747 DATA "COLUMN","PACK","NARROW","PASS","(AGE"
748 DATA "FROM","FLOOR","NEED","(LE","SHARP"
749 DATA "IMPOSSIBLE","ANGLE","IRON","BAR","EMBEDDED"
750 DATA "ONE","CHUTE","ENVELOP","WALK","GLOW"
751 DATA "SMALL","PIECE","SULPHUR","FALL","FROM"
752 DATA "","TIME","TOUCH","DRIP","TREAD"
753 DATA "ACROSS","STRETCH","SMOKE","RING","JET"
754 DATA "START","FURNACE","TEND","MOLTEN","SQUEEZE"
755 DATA "FUME","SWEPT","FIERY","SPARK","PAST"
756 DATA "EAR","SCORCH","SPIRAL","APPARENT","END"
757 DATA "WISP","COME","HOLE","SKY","LADDER"
758 DATA "PLACE","FOOT","FIRST","RUNG","SCREAM"
759 DATA "AGONY","AMIDST","CERULEAN","SWIFT","SWALOW"
760 DATA "ABOUT","TREMBL","MOVEMENT","VIBRATE","MORE"
761 DATA "HEAVEN","STUCK","BANK","STRATO","CUMULUS"
762 DATA "MAGNIFICENT","CHARIOT","FOG","TWENTY","BUFFET"
763 DATA "ROUND","ROOM","BRIDGE","EAGLE","GLINT"
764 DATA "BEAK","VIEW","DESCEND","GET","WIDER"
765 DATA "MILE","MIRROR","CELESTIAL","CIRRUS","SPREAD"
766 DATA "MIST","FLAKE","LEAP","CAN","BARE"
```

```
767 DATA "CRESCENT","STAR","CATHEDRAL","SPUN","CRYSTAL"

768 DATA "EMIT","YELLOW","SPAR","LIGHT","THRONE"

769 DATA "CHILD","BLONDE","CURL","CURVED","MOSAIC"

770 DATA "AMETHYST","POINT","ARCH","CANOP","SILK"

771 DATA "WOMAN","(SIDE"
```

Figure 9.5. Dictionary loader

When the dictionary is held in memory the DATA statements containing all the messages should be typed in and appended to the program 'Encoder' given in Fig. 9.6. This should then be run. It will take each statement in turn, look for each of its words in the dictionary, and code the word into a two-byte representation. The resulting coded DATA statement can then be copied from the screen using the <Copy> key, and the coded statement will replace the literal one. (Make sure that you have kept a copy of your original DATA statements just in case anything goes amiss.) Alternatively, you could adapt 'Encoder' so that it creates a file which *SPOOLs the DATA. Unfortunately the encoding process is so slow and cassettes are so long that you might be better off *SPOOLING each DATA statement as a separate file and then *EXECing them later onto the main program.

```
 10 HIMEM=&5000
 20 *LOAD "DICT" 5000
 30 CLS
 40  REM THIS CODES ADVENTURE TEXT
 50  REM INTO TWO BYTE CODES
 60 code$=""
 70 PRINT"What DATA line to encode?"
 80 INPUT line
 90  RESTORE line
100  READ text$
110 text$=text$+" "
120 CLS
130   PRINT''"SEARCHING....."''
140  REPEAT
150   next_space = INSTR(text$," ")
160    current_word$ = LEFT$(text$,next_space-1)
170   text$=RIGHT$(text$,LEN(text$)-next_space)
180   PROCcode
190     UNTIL LEN(text$)<LEN(current_word$)
200   PRINT ''line;" DATA """;code$;"""."
210   PRINT'"NOW <COPY> THIS DATA LINE"
220 END
```

Figure 9.6 Text encoder (*continues*)

```
230 DEFPROCcode
240 count=&4FFF
250 REPEAT
260   count=count+1
270   find$=$count
280      UNTIL current_word$=find$
290 IF ?count=0 THEN ENDPROC
300 count=count-&4FFF
310  count1=count DIV 64
320  count2=count MOD 64
330       code$=code$+CHR$(35+count1)+CHR$(35+count2)
340 ENDPROC
350    DATA "YOU SEE AN APPLE"
360   REM *****   FROM HERE ON LIST ALL DATA     *****
370   REM *****        TO BE ENCODED.            *****
380   REM *****  COPYING THE PROGRAMS DATA WILL *****
390   REM *****  REPLACE THEM WITH THEIR CODE.   *****
```

Figure 9.6 Text encoder

However, you do not need to do this for The Opal Lily as the encoding process has already been carried out. All you need do is make sure that you type in each of the DATA statements exactly as printed here and that the dictionary has been loaded correctly. Any error at all will create a faulty address which will result at best in a peculiar message and at worst a system crash when the final program is run.

PROCdecode in the main program is called whenever a coded message is to be PRINTed on the screen. The pointer is RESTOREd to the appropriate line and the statement READ in to the variable text$. This is repeatedly done until an empty string is READ in. You will see that every few DATA statements there is such an empty string. These are important as they represent the terminators for that chunk of text.

Each time a DATA line is read in to text$ it is decoded in pairs of characters, controlled by the STEP command in line 1540. The current pair of characters is held in code$ and line 1560 works out the number that code represents by multiplying the ASCII code of the left-hand character by 64 (having subtracted 35) and adding the result to the ASCII code for the right-hand character (having subtracted 36). The subtractions are necessary in order that the characters below ASCII 35 are not used as most of these cannot be printed out and the " character can cause confusing strings.

The resulting number is now used as the address of the required word. Line 1580 is the key line which reads the word from memory. The command "word$ = $number" can be read as "Put in word$ the string which starts at address 'number' and ends with the next

<Return> character found". Before this word is printed on the screen two further tests are made to see (1) whether the word will be split on the edge of the screen when printed and (2) if the leftmost character of the word is '('. The test uses POS, which holds the current horizontal position of the text cursor, and adds it to the length of the current word. If this is greater than the line length then the new line character, CHR$ (13), is printed so that the next word will be on a new line.

If the leftmost character is '(' then it is replaced by CHR$ (127), the backspace character, so that the word part is printed immediately following the start of the word it belongs to. A space is then printed and the whole routine is repeated until the end of text$ is reached. When this happens control is returned to the calling routine and a full stop is printed. This is why the DATA statements are arranged in a slightly irregular manner. As each statement is a full sentence, in some cases a location will have several DATA statements while in others it will have only one.

9.5 Creating a data file

Another way to create the DATA statements for a program such as an adventure is to write a program which works out the DATA for you and sends it to a file on cassette or disk. If we take the objects used in The Opal Lily as an example a program such as OBJWRT2 (Fig. 9.7) will encode the objects as a series of semantic codes. When it is RUN it uses the dictionary file, which it loads, to name each noun in turn and ask the user for its starting room together with various pieces of semantic information. This is encoded into three bytes. The first byte holds the room number (dump1), the second holds four sets of semantic information (dump2), and the third holds additional semantic information as described above (dump3).

```
 10 REM PROGRAM TO PLACE OBJECT CODES IN MEMORY
 20 MODE6
 30 CLS
 40 base=&4C98
 50 HIMEM=base
 60 *LOAD "DICT"
 70 FOR i= 0 TO 86
 80 r=&5170+(i*8)
 90 PRINT "What room is ";$r;" in?"
100 INPUT dump1
110 I$="Is it "
120 dump2=0:dump3=0
130 PRINT I$;" edible?"
140 PROCyesno
```

Figure 9.7 OBJWRT2 (*continues*)

```
150 IF yes=1 THEN dump2=dump2 + 128
    ELSE PRINT I$;"drinkable?":
    PROCyesno:
    IF yes=1 THEN dump2=dump2 + 64
160 PRINT I$;" a container?"
170 PROCyesno
180 IF yes=1 THEN PROCempful
190 PRINT I$;" wearable?"
200 PROCyesno
210 IF yes=1 THEN dump2=dump2+8
220 PRINT I$;" hiding something?"
230 PROCyesno
240 IF yes=1 THEN dump2=dump2+1
250 PRINT I$;" breakable?"
260 PROCyesno
270 IF yes=1 THEN dump3=dump3+64
280 PRINT I$;" mobile?"
290 PROCyesno
300 IF yes=1 THEN dump3=dump3+32
310 PRINT I$;" animate?"
320 PROCyesno
330 IF yes=1 THEN dump3=dump3+16
340 PRINT I$;" burnable?"
350 PROCyesno
360 IF yes=1 THEN dump3=dump3+8
370 PRINT I$;" hidden?"
380 PROCyesno
390 IF yes=1 THEN dump3=dump3+1
400 PROCq(0,dump1)
410 PROCq(1,dump2)
420 PROCq(2,dump3)
430 NEXT
435 PROCsave
440 END
450
460 DEFPROCyesno
470 yes=0
480 REPEAT
490 yn$=GET$
500 yes = INSTR("YNyn",yn$) MOD 2
510 PRINTyes
520 UNTIL INSTR("YNyn",yn$)<>0
530 ENDPROC
540
550 DEFPROCempful
560 PRINT I$;" full?"
570 PROCyesno
580 IF yes<>1 THEN dump2=dump2 +16 :ENDPROC
590 PRINT I$;" open?"
600 PROCyesno
610 IF yes<>1 THEN dump2=dump2 +32:ENDPROC
620 dump2=dump2+48
630ENDPROC
640
650 DEFPROCq(q,u)
```

```
660 ?(base+q+(i*3))=u
670 ENDPROC
680
690 REM READ OBJECT CODES FROM MEMORY
700 REM AND SEND TO A FILE AS DATA STATEMENTS
710 DEFPROCsave
720 PRINT"Please place a blank cassette
    in the recorder"
730 PRINT"Then press Record
    and press any key"
740 g=GET
750 @%=00003
760 *SPOOL "OBJFILE"
770 FOR i=0 TO 86
780 PRINT (11000+i);" DATA ";
790 PRINT STR$(?(base+(i*3)));",";
800 PRINT ?(base+1+(i*3));",";
810 PRINT ?(base+2+(i*3))
820 NEXT
830 *SPOOL
840 ENDPROC
```
Figure 9.7 OBJWRT2

Each time a noun is finished the results are stored in three arrays, called Room%, Stat1%, and Stat2%. When all the information has been coded the second part of the program opens a file called OBJFILE by the command *SPOOL. It then PRINTs on the screen a line of program consisting of a line number, the word DATA, one number from each array, and commas to separate the numbers, giving screen output just like a programmer typing in DATA statements as part of a program, such as:

11001 DATA 12,128,16

As the only information that goes to the file is that which appears on the screen, we have effectively sent a series of program lines to a file on cassette or disk without having to type them in. The second *SPOOL in line 830 closes the file so no extra information can be sent to it by mistake. This file is not the same as a normal BASIC program, however, because BASIC files are held in what is called 'tokenized' form. That is to say a BASIC file would not contain the word DATA held as four bytes (one for each character) but as a one-byte code or token which represents the entire keyword. This is obviously a very efficient way of storing programs both from the point of view of memory and of processing, but it means our screen output, which is only pretending to be the lines of a program, cannot be stored in this way.

The *SPOOL command creates an ASCII file in which every character sent is represented by one byte, so in OBJFILE the word

DATA will be represented by four bytes. It therefore has to be loaded back to the screen in a special way, by typing in direct mode (i.e., not as part of a program) *EXEC "OBJFILE". The file will be called back to the screen just as if you were typing it in, but much more quickly and, what is more, the lines will now be taken into memory as a program just as if you really had typed them in directly. In other words, the file that we have spooled is being treated in exactly the same way as the keyboard. Information is being read in and, if it begins with a number, it is treated as a line of program.

Programs can be merged together using this method (see pages 402 and 403 of the BBC manual and pages 200 and 201 of the Electron manual). You will find that if you had anything in memory when you typed *EXEC "OBJFILE" that the incoming lines have been appended to the existing program. This is obviously useful if it is what you wanted, but frustrating if it corrupts a valuable program you have forgotten to save beforehand. Therefore, always double check when using *EXEC, (1) that, if anything is held in memory, you want it to be combined with the incoming file and (2) that there is no overlap between the line numbers in the programs, because the existing lines will be overwritten by the new ones.

9.6 Saving the data

When you have compiled The Opal Lily's map, object data, and dictionary you can simply load them into memory together and then *SAVE the lot using the small routine below:

```
20 *LOAD "MAP"
30 *LOAD "DICT"
40 PRINT "NOW CHANGE TO A BLANK CASSETTE THEN
PRESS ANY KEY"
50 G=GET
60 *SAVE "OPALDAT" 4C97 5FFF
```

You should have all the previously saved data blocks separately held on one cassette in the order "MAP" and "DICT". These will be loaded in turn and held in RAM next to each other, provided they have been *SAVEd correctly. When the program asks for a blank cassette insert a new one so that you have the complete "OPAL" file on a separate tape. Otherwise you will have difficulty finding it again. When you have typed in the listing for The Opal Lily this tape will be your data tape used by the main program.

9.7 Text handling

Text handling in adventure programs can be very sophisticated provided you can put up with the (by now familiar) problems of memory shortage and complexity of algorithm. The Opal Lily (Fig. 9.8) does not represent a great advance on the normal two-word adventure input routines but I have included one or two improvements to show how you can go about achieving a complex text parser. (A parser is simply a program for analysing sentences into their component parts.) The key features of the input routine are:

1. Input can be in lower or upper case.
2. Extraneous spaces, numbers, and other characters are removed so some common typing mistakes do not mean that the player has to retype a command.
3. Provided the player types the first letter of each word correctly the machine will guess the intended word if it was mistyped. Input can be two words, several words, single words, or single letters (in some cases).
5. The input checking routine is very fast.
6. One conjunction and one preposition are allowed for complexity of input.

Many adventures have none of these features. Some programmers seem to think that because input is not actually part of the game structure players will put up with all sorts of quirks in the input system. However, the so-called ‘user interface’ for adventures is crucial to enjoying the game. If processing is slow, if mistakes are easy to make, if easy forms of input are prevented, the player may become fed up with playing even if the actual game is quite original.

You will see that the total input routine is one of the longest in the program, so let us go through procedure by procedure to see exactly what is going on and what could be developed further.

The main loop contains PROCcommand, which asks for and decodes input, and then a GOSUB call dependent on v%, the variable representing the input verb. PROCcommand firstly looks to see if any text remains to be processed (line 1045). This is indicated by the variable conj, meaning conjunction, being set to 1. If it is set then the string held as temp$ is processed. Temp$ holds any text left over from the previous processing cycle. If it is not set then PROCinput is called.

PROCinput does two things. Firstly, it gets the text using INPUT LINE to make sure that no errors are generated from faulty punctuation. Then it looks at each character in the input string in turn and turns it to upper case if it is in lower case and throws it away if it is anything other than an upper case letter or a space. This is done simply by checking the ASCII codes of the input text and altering each character accordingly. Finally, it reduces any strings of multiple spaces (caused, for example, by the player pressing the space bar for too long or by the routine deleting punctuation between two spaces) to a single space. This last stage is important because, as we have seen in Chapter 7, the space is used as a marker for word endings. If we have more spaces than words then the rest of the decoding routines will create errors.

At this point we now have a string called text$ which either has just been typed in or was left over from the previous input cycle. The program next looks to see if the command was only one word. This will be the case if there is no space in the input string. If it is, PROCone_word is called. PROCone_word firstly makes sure that the word is allowed on its own. Allowed commands include the six possible directions, Inventory and Quit. It does this by looking only at the left-most letter so other single words with the same initial letters would be treated identically. For example, if SWEAR was typed in the program would treat it as SOUTH. This is done in order that single-letter and single-word commands can be dealt with by the same routine. If we wanted to allow single-letter commands but prevent confusion between words with the same initial letters we would have to use two procedures and add an extra test.

The left-most letter is compared with a string called dir$ (for 'direction'). If there is no match then PROCcommand ends and another input is requested. If the command is found then either it will be a direction or it will be something else. If it is a direction then it can be treated as if it was the two-word command "GO (direction)" so the verb variable is set to 7 (for 'GO') and the noun variable to the direction number, dependent on the letter's position in dir$. You will see that the GO subroutine deals both with single-letter and multi-word instructions.

If the command is not a direction then the verb is set to a number beween 1 and 26 derived from its ASCII code (line 1720). In other words, the only allowed single-word commands are the first 26 in the verb dictionary, one for each letter. If you wish to use this method in another program it is a good idea to place the most common verbs in these slots in your dictionary.

If there are spaces in text$ then it consists of more than one word. The process is therefore like the two-word decoding described in Chapter 7. Firstly, the words are checked to see if they are legal; if they are then control variables are set to their values. All the characters up to the first space are stripped off text$ and treated as the first word, which has to be a verb to be legal. So PROCword is used to check that it is an allowed verb. PROCword is also used to check the nouns when found. Consequently, it needs a set of parameters telling it where to begin the search in the dictionary and the length of the strings it is meant to be looking at. You will remember that the data structures are different for nouns and verbs, the former being eight characters long, the latter only seven. PROCword returns a variable called 'number'. If number is 0 then this flags an error. If it is greater than 0 then this is the control variable and either v% (for verb), n1% (for the first noun), or n2% (for the second noun) will be set to it.

PROCword compares the input word with a subset of the dictionary stored above HIMEM. It is at this point that input routines usually slow up because many words have to be checked before a match is found. However, the data structure we have used means that very few comparisons have to be made—a maximum of ten and an average of about four. This is because the procedure knows exactly where to look for likely candidates. Its first choice is the first word in the appropriate section of the dictionary which begins with the same letter as the input word (found by multiplying the ASCII code of the leftmost letter of the word by the allowed word length and adding the result to the address of the start of that portion of the dictionary). If there is no match then the entry for that word contains a pointer to the next word beginning with the same letter. If that does not match then it has a pointer to the next word. This continues until all the words in the verb dictionary or the noun dictionary which begin with the correct letter have been checked. If no match is found then the last entry in the list will have a zero pointer which tells the searching routine that no match exists and it ends.

If a match is found then the routine terminates and the variable number is set to the word's number in the list. If a partial match is found (i.e., if a word is found which begins with the same letter as the input word but otherwise differs) then PROCspel is called. PROCspel simply compares the current partial match with the input word to see how many letters the former has that correspond to letters in the latter. This is not a particularly sophisticated comparison and you might like to improve it, but given the rather limited dictionary used by adventures it works well enough. It simply uses INSTR to count the number of letters in the partial match which are also in the input word. It does not, for example, attempt to ensure that it does not count a letter twice; nor does it try to match position as well as character type. Consequently, if you typed FILD by mistake for FILE the program will find the best match as FIELD because there are more letters in common even though their positions are different.

PROCspel sets a variable called best_score to the highest number of matches it finds and keeps in poss_count the address of that match in memory. Then if PROCword fails to find a perfect match for the input word the variable 'number' is set to 0 and PROCerror is called. If no match at all has been found then the input word is called a mistake and the player is told as much. However, there is usually a partial match, in which case the match held at poss_count will be printed with a request for the user to say whether it is a correct guess or not. In this way a large number of spelling mistakes and slips of the fingers are compensated for, saving a great deal of mistyping. I find that especially in the process of debugging the program this routine saves a great deal of frustration.

If the user types 'N' (for 'No') then the whole command is abandoned and another command is requested. This spelling checker has one major flaw, however, if you want to use it in a situation where cheating is to be prevented. A player will very quickly realize that all that is needed to discover a fair percentage of the vocabulary of the game is to keep typing in errors and record the suggested matches that the program comes up with. Whether you regard this as a flaw or not depends on your point of view. Obviously there are so many opportunities for you to cheat, because you have the whole program laid out before you in the book, that a routine like this does not really matter. In addition, there are some games which give you all the vocabulary beforehand. In other games a tried and trusted technique of players is to keep trying words randomly until something happens, so this could be regarded as an extension of that method of 'solution'. And knowing the words does not necessarily tell you how to use them. However, if you did want to prevent this sort of cheating you would either have to remove the spelling checker or you would have to add a routine which monitored the words that the player was allowed to know.

This kind of routine would not be too difficult to write. It would only be necessary to set aside one bit of memory for each word. This could easily be combined with the actual dictionary by using the left-most (most significant) bit of the byte holding the first letter of each word. (As capital letter character codes do not go above 90 this bit is not used but you will have to change all the text decoding routines accordingly.) If the bit is set then that word has not yet been 'found' by the player so cannot be used by the spelling checker. However, when the player finds the word (e.g., by using the TAKE routine or entering a location which uses the word in its description) the bit will be 'unset' or 'reset' and the spelling checker can use it.

If a legal verb is found then the next word is stripped from text$ and treated as a noun. PROCword is called with different parameters and the same series of tests is made. At this point one could add a slight degree of sophistication to the routine by allowing articles (i.e., 'the' and 'a'). All that would be required is a test to see if the current word is either of these and if it is it is thrown away and the next word from text$ is taken. The noun is either the leftmost word in text$ or the whole of text$ if there are no spaces. Consequently if a space is missed out in typing an error will probably result here, though is some cases the spelling checker will still find a match.

At this point we have described a reasonably intelligent two-word command system. However, to make a fully fledged parser which can deal with many different kinds of English input more work is obviously required. Therefore I have added two slight variations to show the way that such a routine might be developed. They are not called if either one- or two-word input is used, but only if there is more text to process after v% and n1% have been assigned.

The first of these is the conjunction "AND". This conjunction can actually mean several different things but this program only allows one of those meanings, namely, linking two commands together. For example, "TAKE ROCK AND GO NORTH" is allowed, but "TAKE ROCK AND SWORD" is not because the second verb is missing. The two commands must be two-word commands (or more) and must be complete. If AND is found in text$ then it is stripped off, the variable conj is set to 1, the whole of text$ is put into temp$, and text$ is wiped clean. Then the procedure ends and the command that has so far been decoded is executed. However, when that command has been completed and PROCcommand is called again new input will not be requested but the text stored in temp$ will be placed back into text$ and this will be decoded instead. Needless to say, several commands can be embedded in one input string provided they are all linked by AND—up to the string-handling limit of the micro.

Some other conjunctions could be treated in a similar way. THEN, for example, could be treated exactly like AND. However, others would require more thought. For example, use of AFTER (apart from requiring different verb forms) would mean that the first part of the command should be processed second.

If AND is not found then the program looks for the preposition WITH. It will continue to do so until it is found or until all the words have been removed from text$. If WITH is found then it will also look for a noun, again using PROCword, setting n2% to the number of any noun found. It also sets the variable prep to 1 as some routines can make sense with or without the preposition.

Using prepositions allows a large increase in the potential of an adventure though this will also involve extra coding in the verb routines. In effect every new preposition allowed by our program adds 1 to the available classes of noun. Without prepositions we have just the object, the thing acted upon by the verb. With the preposition WITH we have introduced the category 'instrument', i.e., things that can be used for other things. For example, there is a world of difference between "FIGHT DRAGON", "FIGHT DRAGON WITH SWORD" and "FIGHT DRAGON WITH BANANA". (Remember it would also be possible to allow commands like "FIGHT DRAGON WITH CAUTION".)

Other prepositions allow other kinds of noun category. TO allows the category 'destination', i.e., the thing which something is directed towards, as in "GIVE BANANA TO DRAGON" or "THROW DWARF TO LIONS". FROM allows the reverse of destination, the 'source'; ON, IN, UNDER, OVER, and so on, allow complex positions; FOR allows specification of duration; and BY allows specification of a conveyance of some kind. Consider the apparent complexity of a command like "SEND THE DWARF FROM THE CAVE TO THE LIONS BY CHARIOT WITH A GUARD", for example. This kind of command could be decoded in exactly the same way as our single example, by decoding each preposition in turn. However, as I have remarked before, if you wish to do this in your adventures, something else will have to be sacrificed.

9.8 Verb routines

Any adventure of this type is really a large database with a series of routines for manipulating the data held within it. So far we have examined the nature and structure of the database and of the input controls which determine which manipulating routines are called. Let us now examine the verb routines, which are the main routines whereby the data can be manipulated.

In essence all the verb routines in The Opal Lily have the same structure, and this is true for a large majority of adventures. They consist of the following formula:

> Check that the input command makes sense in the current location and, if it does, alter either the map data or the object data, or print a rewarding message; otherwise print a different message indicating an error.

Verb routines really only call one or more of three kinds of routine:

a routine to affect map data, a routine to affect object data, or a routine to print a new message. However, there are different kinds of map and object data and different kinds of message, so we will probably not be able to make do with just three routines, even if they can take different parameters, though we might be able to reduce the actual number of routines well below the number of verbs multiplied by the number of nouns. In addition, we need to include tests to ensure that the input command makes sense in a given location, which will add another routine or set of routines to the list. If we list the key variables in The Opal Lily we will get some idea of the kind of tests and the kind of alterations that might be needed.

These main variables are:

R%	the player's current location
Object byte 1	the location of a given object
Object byte 2	four kinds of status information on a given object
Object byte 3	four more kinds of status information on a given object
Map byte 1	the availability of a given exit from the current location

This is the entirety of the significant control variables in the game, not a large number. The input commands can therefore only have one of the following effects, if we discount one or two special verbs like "INVENTORY":

1. End the game (either by voluntary quitting or by 'dying').
2. Change R% (i.e., move to a different location, which could be voluntary or involuntary movement).
3. Change the location of an object by taking it (setting the relevant variable to 99), dropping it (setting the variable to R%), or destroying it (which might be deliberate or the consequence of some other action, but results in the variable being set to 0).
4. Change one or more of the status bits by opening, burning, breaking, etc.
5. Reveal an exit, i.e., set the 'hidden exit' bit to 0.
6. Some combination of the other five, either by repeating an action or combining two or more actions.

It looks as if we should be able to code the effects of all verb routines using just five routines, therefore, though some verbs may require a sequence of such routines. Could a similar thing be done for the tests? A list of all possible tests would be:

1. Is the location suitable for the desired action?
2. Is the main object at this location?
3. Is the instrument (the auxiliary object) at this location?
4. Are the appropriate status bits set to the correct values?

It would appear that five routines or functions would be sufficient to perform these tests, though we may want more than one test performed at a given location. We thus have a set of five WRITE routines which will write particular data to the database and a set of five READ routines which will read the current contents of the database. Provided we use the correct combination of these we should be able to carry out all the necessary verb manipulations. A slightly more specific model of the typical verb routine would thus be:

1. Perform all necessary tests by using the READ routines.
2. If at least one test is negative then print an error message and end the verb routine; otherwise proceed to the next stage.
3. Perform all the required WRITE procedures to alter the database.
4. Print the appropriate discovery messages.
5. End the verb routine.

There might also be some exceptional verbs which control other variables or automatically carry out operations without such tests, like the QUIT and INV verbs, but the majority of verbs should fit within this general pattern.

The key feature of these routines has to be their flexibility. Different verbs will require tests of different bytes and for different values. Consequently, the READ and WRITE routines will need several parameters which the calling verb will have to pass values to. The general form for a verb subroutine would therefore be:

```
IF FNtest(a,b,c...n) THEN PROCact(o,p,q...z)
```

where the brackets contain the values to be passed as the parameters of the functions or procedures. FNtest could, for example, read the byte which was b bytes from byte a and compare it with c, while PROCact could add value p to the value in the byte (q*r) above byte s. The unfortunate thing is that we have to design these functions and procedures to use only and all the parameters given, and thus we have to send values to each parameter even if they are not to be used. Our program therefore uses several such routines and only calls the appropriate ones in order to balance the efficiency of using one procedure for several verbs against the inefficiency of having to pass redundant parameters.

NOTES

(a) The listing will not run if typed in in this form. All unnecessary spaces and all REM statements must be removed. Where possible lines must be converted to multi-statment lines. (This will also increase speed of execution). Shortening variable names will also save space.

(b) All DATA must be typed in exactly as printed here or the decoding routines will not run accurately. If you find that a message is odd it will probably be because of mistyping the DATA.

(c) Do not put the DATA statements on multi-statement lines as this will produce ´No such line´ errors when RESTORE is used.

```
10 MODE 6
11 HIMEM = &4FFF
20 PROCst
21 PROCds
28
29  REM MAIN LOOP
30 REPEAT
31   v% = 0
32   n1% = 0
33   n2% = 0
34   PROCcommand
40   IF v%<>0 THEN GOSUB (4000+(50*v%))
50   PROCaf
60   UNTIL d% = 1 OR qu
68
```

Figure 9.8 The Opal Lily (*continues*)

```
 69  REM END ROUTINE
 70 PROCdd
 80 END
 88
 89  REM DISPLAY ROUTINE
 90 DEF PROCds
100 CLS
101 K% = 0
102 PRINT
103 RESTORE (9101+(R%*10))
109  REM PRINT EACH SENTENCE OF DESCRIPTION
110 REPEAT
111   READ text$
112   PROCdecode
113      IF text$<>"" THEN PRINT CHR$(8);"."
114   UNTIL text$ = ""
120 PRINT CHR$(11);
121 PRINT
122 PROCmp(R%)
123 PRINT
124 vp = VPOS
125 PRINT C$
129  REM DISPLAY ANY OBJECTS AT THIS LOCATION
130 FOR k = 1 TO 83
140   IF FNo(0,k) = R% OR (( FNo(2,k) AND 1) =
1) PROCrev
160   NEXT
170 IF vp+1 = VPOS THEN PRINT TAB(9, VPOS-1);N$
180 ENDPROC
188
189  REM DISPLAY ANY AVAILABLE EXITS
220 DEF PROCmp(I)
221 K% = 0
240 pk = FNex(3)
245 IF pk AND 64 THEN 310
250 FOR i = 0 TO 5
251   IF pk AND (2^i) PRINT P$;
MID$(ns$,(i*5)+1,5)
300   NEXT
310 IF FNex(1) = 0 ENDPROC
320 K% = K%+( FNex(1))*3
330 GOTO 240
340 ENDPROC
348
349  REM MOVE LOCATION IF EXIT IS OKAY
350 DEF PROClk
355 LOCAL k
360 k = 0
361 REPEAT
362   K% = K%+k
363   k = FNex(1)*3
364   UNTIL FNex(3) AND g OR FNex(1) = 0
390 IF FNex(3) AND 64 THEN ENDPROC
```

```
  400 IF FNex(3) AND g THEN R% = ?(H-2+(R%*3)+K%)
  410 ENDPROC
  418
  420 DEF FNex(byt) = ?(H-byt+(R%*3)+K%)
  428
  429  REM SPECIAL MOVEMENT CHECK
  430 DEF PROCch(A%,number,C%)
  440 K% = 0
  441 IF nl%<>A% OR R%<>number THEN ENDPROC
  450 IF FNex(3) AND 64 THEN ?(H-3+(R%*3)+K%) =
?(H-3+(R%*3)+K%)-64
  460 IF FNex(1) = 0 THEN RESTORE
(12000+(v%*100)+nl%) : READ text$ : CLS :
PROCdecode : PROCpa : v% = 7 : n% = C% : GOSUB
4350 : ENDPROC
  470 K% = K%+( FNex(1))*3
  471 GOTO 450
  490 ENDPROC
 1038
 1039  REM DECODE INPUT COMMANDS
 1040 DEF PROCcommand
 1045 IF conj = 1 THEN text$ = temp$ ELSE
PROCinput
 1050 conj = 0
 1051 sp = INSTR(text$," ")
 1052 IF sp = 0 THEN vr$ = text$ : PROCone_word :
ENDPROC
 1065 vr$ = LEFT$(text$,sp-1)
 1066 text$ = RIGHT$(text$, LEN(text$)-sp)
 1067 IF LEN(vr$)>5 THEN vr$ = LEFT$(vr$,5)
 1070 PROCword(vr$,ve,7)
 1071 IF number = 0 THEN PROCer : IF number = 0
THEN ENDPROC
 1080 v% = number
 1081 sp = INSTR(text$," ")
 1082 IF sp<>0 THEN n$ = LEFT$(text$,sp-1) ELSE n$
= text$
 1085 IF LEN(n$)>6 THEN n$ = LEFT$(n$,6)
 1090 IF RIGHT$(n$,1) = "S" THEN n$ = LEFT$(n$,
LEN(n$)-1)
 1095 IF text$<>n$ THEN text$ = RIGHT$(text$,
LEN(text$)-sp) ELSE text$ = ""
 1100 PROCword(n$,nn,8)
 1101 IF number = 0 THEN PROCer : IF number = 0
THEN ENDPROC
 1110 nl% = number
 1111 IF text$ = "" THEN ENDPROC
 1120 REPEAT
 1121    IF LEFT$(text$,3) = "AND" THEN conj = 1 :
E$ = "AND" : temp$ = RIGHT$(text$, LEN(text$)-4) :
text$ = "" : GOTO 1165
 1130    IF LEFT$(text$,4) = "WITH" THEN prep = 1 :
text$ = RIGHT$(text$, LEN(text$)-5)
 1135    sp = INSTR(text$," ")
```

Figure 9.8 (*continues*)

```
 1136   IF sp<>0 THEN n$ = LEFT$(text$,sp-1) ELSE
n$ = text$
 1141   IF text$<>n$ THEN text$ = RIGHT$(text$,
LEN(text$)-sp) ELSE text$ = ""
 1150   IF n$<>"" THEN PROCword(n$,nn,8)
 1155   n2% = number
 1165   UNTIL text$ = "" OR (sp = 0 AND number =
0)
 1170 ENDPROC
 1173
 1174  REM INPUT ROUTINE
 1175 DEF PROCinput
 1200 REPEAT
 1201   PRINT
 1202   INPUT LINE "WHAT NOW",text$
 1203   UNTIL text$<>""
 1220 CLS
 1221 i = 0
 1222 REPEAT
 1223   i = i+1
 1224   l% = FNt(i)
 1245   IF l%>90 THEN text$ = FNl(i-1)+
CHR$(l%-32)+ FNr
 1255   IF FNt(i)<65 AND FNt(i)<>32 THEN text$ =
FNl(i-1)+ FNr : i = i-1
 1265   IF FNt(i) = 32 AND FNt(i+1) = 32 THEN
text$ = FNl(i-1)+ FNr : i = i-1
 1270   UNTIL i = LEN(text$)
 1280 ENDPROC
 1283
 1284   REM SUNDRY TEXT SPLITTING FUNCTIONS
 1285 DEF FNt(t) = ASC( MID$(text$,t,1))
 1290 DEF FNr = RIGHT$(text$, LEN(text$)-i)
 1295 DEF FNl(l) = LEFT$(text$,l)
 1298
 1299  REM DECODE WORDS
 1300 DEF PROCword(word$,c%,T%)
 1310 E$ = word$
 1311 best_score = 0
 1312 E$ = LEFT$(E$, INSTR(E$," ")-1)
 1313 k$ = LEFT$(E$,1)
 1314 pt = 0
 1315 pn% = 0
 1316 K% = c%+(( ASC(k$)-65)*T%)
 1345 REPEAT
 1346   K% = K%+(pt*T%)
 1347   pt = ?(K%+T%-1)
 1348   IF E$<>$K% THEN PROCspel
 1365   UNTIL pt = 0 OR pt = 79 OR E$ = $K%
 1370 IF E$ = $K% THEN number = ((K%-c%)/T%)+1
ELSE number = 0 : pn% = ((poss_count-c%)/T%)+1
 1380 ENDPROC
 1383
 1384  REM INFORM PLAYER OF ERRORS
 1385 DEF PROCer
 1386 IF pn%<1 THEN PRINT text$;" IS A MISTAKE" :
ENDPROC
 1390 PRINT "DID YOU MEAN ";$poss_count;" (Y/N)?"
 1395 yn$ = GET$
 1396 A% = INSTR("yYnN",yn$)
 1397 IF A% = 0 THEN 1395 : IF A%>2 THEN ENDPROC
 1410 number = pn%
 1411 ENDPROC
 1413
 1414  REM FIND THE BEST MATCH FOR AN INPUT WORD
 1415 DEF PROCspel
 1420 sc = 0
 1421 FOR i = 1 TO LEN(E$)
 1422   IF INSTR($K%, MID$(E$,i,1))<>0 THEN sc =
sc+1
 1430   NEXT
 1440 IF sc>best_score THEN best_score = sc :
poss_count = K%
 1445 ENDPROC
 1448
 1449  REM DECODING THE COMPRESSED TEXT
 1520 DEF PROCdecode
 1521 j = 0
 1522 number = 0
 1523 IF text$ = "" THEN ENDPROC
 1540 FOR i = 1 TO LEN(text$) STEP 2
 1541   code$ = MID$(text$,i,2)
 1560   number = (( ASC( LEFT$(code$,1))-35)*64)+(
ASC( RIGHT$(code$,1))-36)
 1580   word$ = $(ve+number)
 1585   j = LEN(word$)+1
 1590   IF POS+j>36 THEN PRINT CHR$(13)
 1595   IF LEFT$(word$,1) = "(" THEN word$ =
CHR$(127)+ RIGHT$(word$, LEN(word$)-1)
 1600   PRINT word$;" ";
 1601   NEXT i
 1620 ENDPROC
 1628
 1629  REM SPECIAL ROUTINE FOR SINGLE WORD
COMMANDS
 1630 DEF PROCone_word
 1631 n% = 0
 1632 v% = INSTR(dir$, FN1(1))
 1633 IF v% = 0 THEN PROCer : ENDPROC
 1700 IF v%<7 THEN n% = v% : v% = 7 : ENDPROC
 1720 v% = ASC( FN1(1))-64
 1721 ENDPROC
 1998
 1999  REM PRINT CURRENT VERB
 2000 DEF FNvb
 2010 j$ = $(ve+(v%*7)-7)
 2020 = j$
```

Figure 9.8 (*continues*)

```
 2098
 2099  REM "YOU VERB IT"
 2100 DEF PROCit
 2101 PRINT Z$; FNvb;" ";
 2102 PROCn
 2103 ENDPROC
 2138
 2139  REM "YOU WANT TO VERB IT"
 2140 DEF PROCtry
 2141 PRINT Z$;"WANT TO "; FNvb;" ";
 2142 PROCn
 2143 PRINT w$
 2144 ENDPROC
 2198
 2199  REM CHECK OBJECT IS NOT HIDDEN
 2200 DEF PROCrev
 2210 IF ( NOT(( FNo(2,k) AND 1))) = TRUE THEN
PROCdis : ENDPROC
 2220 IF FNo(1, FNo(0,k)) AND 1 THEN ENDPROC
 2230 IF FNo(0, FNo(0,k)) = R% THEN PROCdis : nl%
= k : ? FNin = R% : PROCoin(2,-1)
 2240 ENDPROC
 2298
 2299  REM DISPLAY OBJECTS
 2300 DEF PROCdis
 2301 RESTORE (19999+k)
 2302 READ text$
 2303 PROCdecode
 2304 PRINT
 2305 ENDPROC
 2398
 2399  REM DISCOVER HIDDEN OBJECTS BY LOOKING
 2400 DEF PROCas
 2401 FOR lm = 1 TO 87
 2402   IF ( FNo(0,lm) = nl%) AND (( FNo(2,lm) AND
1) = 1) THEN ?(O%+((lm-1)*3)) = 99 : RESTORE
(19999+lm) : READ text$ : PRINT C$; : PROCdecode
 2430   NEXT
 2431 ENDPROC
 2998
 2999  REM CHECK FOR CONDITIONS AFTER EACH MOVE
 3000 DEF PROCaf
 3001 IF R%>28 AND R%<41 THEN oxy = oxy-1 : IF
oxy<0 THEN PRINT "YOUR AIR RUNS OUT." : d% = 1
 3025 IF R%>29 AND R%<41 AND (( FNo(2,54) AND 2) =
2) THEN PRINT "THE MUD IS WASHED AWAY" : nl% = 54
: ? FNin = 24 : PROCoin(2,-2)
 3030 IF R% = 8 AND ( NOT( FNo(2,54) AND 2)) =
TRUE AND L%<1 THEN PRINT R$;"STUNG BY WASPS" : d%
= 1
 3040 IF (?(O%+184) AND 4)<>4 AND R% = 46 THEN
PRINT Z$;"FALL OVER AND BREAK YOUR NECK" : d% = 1
 3050 IF R% = 54 AND FNo(0,52) = 99 THEN PRINT
```

```
R$;"PULLED UPWARD" : R% = 53
 3060 IF R% = 22 AND FNo(0,55)<>99 THEN PRINT "THE
WITCH DOCTOR SEES ";R$;"ALONE AND WAVES HIS FAN."
: d% = 1
 3070 IF R% = 55 AND (?(O%+34) AND 4)<>4 THEN
PRINT R$;"SCALDED TO DEATH" : d% = 1
 3080 IF X% = 1 AND R% = 45 PRINT "A VAMPIRE RISES
AND NIBBLES YOUR NECK" : d% = 1
 3090 IF R% = 37 AND ?(O%+45) = 99 THEN nl% = 14 :
v% = 7 : PROCch(nl%,37,5)
 3100 IF R% = 21 THEN ?(O%+51) = 0
 3110 IF R% = 59 AND J%<>1 THEN PRINT R$;"LOST IN
SMOKE"
 3120 IF R% = 60 AND E% = 0 THEN PRINT R$;"TRAPPED
BY A RING OF FIRE"
 3130 IF R% = 79 AND ?(O%+216) = 99 THEN nl% = 73
: v% = 38 : PROCch(nl%,79,3)
 3190 ENDPROC
 3298
 3299  REM END OF GAME
 3300 DEF PROCdd
 3310 IF d% = 1 PRINT R$;"DEAD"
 3320 IF L% = 2 THEN PRINT "WELL DONE!"
 3390 ENDPROC
 3998
 3999  REM PAUSE
 4000 DEF PROCpa
 4001 TIME = 0
 4002 REPEAT
 4003   UNTIL TIME = 400
 4004 ENDPROC
 4047
 4048  REM VERBS BEGIN HERE
 4049  REM ASK
 4050 PRINT "I´M A STRANGER HERE MYSELF"
 4051 RETURN
 4098
 4099  REM BREAK
 4100 PROCh(nl%)
 4101 IF h% = 0 THEN RETURN
 4115 IF FNb(2,64) THEN PROCit : PROCoin(2,64) :
RETURN
 4120 IF FNb(2,128) THEN PRINT l$;"BROKEN" :
RETURN
 4130 PRINT K$;"BREAK IT"
 4131 RETURN
 4148
 4149  REM CLIMB
 4150 IF nl%<>21 AND nl%<>4 THEN PRINT H$ : RETURN
 4156 IF FNo(2,49) AND 2 THEN PROCch(21,13,5) :
RETURN
 4160 v% = 7
 4161 GOSUB 4350
 4162 RETURN
 4198
```
Figure 9.8 *(continues)*

```
 4199  REM DROP
 4200 IF FNg THEN PRINT y$ : RETURN
 4220 ? FNin = R%
 4221 PROCit
 4222 D% = D%+1
 4240 IF R%>65 THEN PRINT "IT FALLS THROUGH THE
CLOUD" : ? FNin = 0 : RETURN
 4245 IF D%>12 THEN r = RND(3) : IF r = 1 AND
FNb(2,64) = 64 THEN PRINT "YOUR CLUMSINESS HAS
BROKEN IT" : ? FNin = 0
 4246 RETURN
 4248
 4249  REM ENTER
 4250 PROCch(45,1,2)
 4251 PROCch(53,1,2)
 4252 PROCch(45,61,5)
 4253 PROCch(82,15,4)
 4254 PROCch(47,65,6)
 4255 RETURN
 4298
 4299   REM FILL
 4300 IF FNg THEN PRINT y$ : RETURN
 4320 IF FNb(1,16) THEN PROCit : PROCoin(1,32) :
RETURN
 4330 IF FNb(1,32) THEN PRINT Q$ : RETURN
 4340 IF FNb(1,48) THEN PRINT l$;"FULL"
 4348 RETURN
 4349  REM GO
 4350 IF n% = 0 THEN nl% = INSTR(dir$,
CHR$(nl%+64)) ELSE nl% = n%
 4355 IF nl%>6 THEN PRINT "NO WAY!" : RETURN
 4360 g = 2^(6-nl%)
 4361 K% = 0
 4362 PROClk
 4363 PROCds
 4364 RETURN
 4398
 4399  REM HOLD
 4400 IF (nl% = 30 OR nl% = 35) AND (R%<30 OR
R%>40) THEN oxy = 12 : PRINT Z$;"FILL YOUR LUNGS"
: ELSE PRINT K$
 4445 RETURN
 4448
 4449  REM INVENTORY
 4450 PRINT U$;
 4451 vp = VPOS
 4452 FOR k = 1 TO 87
 4453   IF FNo(0,k) = 99 THEN PROCdis
 4490   NEXT
 4495 IF vp = VPOS THEN PRINT N$
 4496 RETURN
 4498
 4499  REM JUMP
 4500 IF R% = 66 THEN PRINT "DOWN YOU GO" : R% =
10 : RETURN
 4530 PRINT "WHAT FUN!"
 4531 RETURN
 4548
 4549  REM KILL
 4550 PRINT R$;"TOO WEAK"
 4551 RETURN
 4558
 4559  REM LOOK
 4600 IF n1% = 0 THEN PROCds : RETURN
 4605 PROCh(n1%)
 4606 IF h% = 0 THEN RETURN
 4610 IF FNb(1,1)<>1 THEN PRINT C$;N$ : RETURN
 4620 PROCoin(1,-1)
 4621 IF FNo(0,n1%) = R% THEN PROCds ELSE PROCas
 4640 RETURN
 4648
 4649  REM MOVE
 4650 GOSUB 5000
 4700 RETURN
 4748
 4749  REM OPEN
 4750 IF FNg AND ? FNin<>R% THEN PRINT y$ : RETURN
 4754 IF FNb(1,16) = 16 THEN PRINT 1$;"OPEN" :
RETURN
 4755 IF n1% = 3 AND n2% = 0 AND FNb(1,16)<>16
THEN PROCtry : RETURN
 4756 IF n1% = 26 THEN PRINT R$;"BOWLED OVER BY AN
ANGRY GALE" : PROCpa : ?(O%+75) = 0 :
PROCch(n1%,59,5) : J% = R%-58 : RETURN
 4760 IF n1% = 3 AND (n2%<>11 OR ?(O%+30)<>99)
THEN PRINT "THAT WON´T WORK" : RETURN
 4780 IF FNb(1,32) = 32 THEN PROCit :
PROCoin(1,16) : ELSE PRINT K$;"OPEN IT"
 4785 IF R% = 45 THEN X% = 1
 4795 RETURN
 4798
 4799  REM PLANT
 4800 PROCh(n1%)
 4801 IF h% = 0 THEN RETURN
 4830  IF n1% = 61 OR n1% = 20 THEN PRINT "WORMS
EAT IT" : ?(O%+180) = 0 : ENDPROC
 4847 RETURN
 4848
 4849  REM QUIT
 4850 qu = TRUE
 4851 PRINT "SAVE GAME?"
 4885 REPEAT
 4886   g$ = GET$
 4887   UNTIL INSTR("YyNn",g$)<>0
 4890 IF INSTR("YyNn",g$)<3 THEN PROCsa
 4895 RETURN
 4898
```

Figure 9.8 *(continues)*

```
 4899  REM REAP
 4900 PROCch(72,75,1)
 4901 PROCch(87,75,1)
 4902 RETURN
 4948
 4949  REM SHUT
 4950 IF FNg AND ? FNin<>R% THEN PRINT y$ : RETURN
 4965 IF FNb(1,16) = 16 THEN PROCit :
PROCoin(1,-16) : ELSE PRINT 1$;"SHUT"
 4980 RETURN
 4998
 4999  REM TAKE
 5000 IF nl% = 30 OR nl% = 35 THEN GOSUB 4400 :
RETURN
 5005 IF NOT( FNg) THEN PRINT 1$;"YOURS" : RETURN
 5010 IF ( FNo(0,nl%)<>R% AND FNo(0,
FNo(0,nl%))<>R%) OR (( FNo(2,nl%) AND 1) = 1) THEN
PRINT I$;"NOT HERE" : RETURN
 5015 IF nl% = 55 THEN PRINT "HE´S NOT THAT EASY
TO PERSUADE" : RETURN
 5016 IF nl% = 74 AND n2%<>65 THEN PROCtry :
RETURN
 5020 ? FNin = 99
 5021 PRINT "NOW ";U$;
 5022 k = nl%
 5023 PROCdis
 5025 IF FNo(2,nl%) AND 1 THEN PROCoin(2,-1)
 5047  RETURN
 5048
 5049  REM UNTIE
 5050 PROCh(nl%)
 5051 IF FNg AND h% = 0 THEN PRINT y$ : RETURN
 5060 PROCit
 5061 ?(O%+48) = 0
 5062 ?(O%+63) = 99
 5063 ?( FNin+2) = 0
 5100 RETURN
 5148
 5149  REM WEAR
 5150 IF nl% = 54 GOSUB 6350 : RETURN
 5160 PROCh(nl%)
 5161 IF h% = 0 THEN RETURN
 5170 IF FNb(1,4) THEN PRINT 1$;"BEING WORN" :
RETURN
 5175 IF ( FNo(1,nl%) AND 8)<>8 THEN PRINT S$ :
RETURN
 5176 PROCoin(1,-4)
 5177 ? FNin = 99
 5178 PROCit
 5182 IF nl% = 43 THEN ? FNin = 0 : nl% = 62 : ?
FNin = 99 : PROCoin(1,-4) : st = 1
 5190 IF nl% = 51 OR nl% = 25 THEN oxy = 500 :
PRINT Z$;"BREATHE CLEAN AIR"
```

```
 5195 RETURN
 5198
 5199  REM BRIBE
 5200 IF n2% = 0 THEN PROCtry : RETURN
 5210 IF n1% = 55 AND n2% = 27 AND FNo(0,55) = R%
THEN PRINT "THE NATIVE GOES WITH YOU" : ?(O%+162)
= 99 : ?(O%+78) = 0 : RETURN
 5247  RETURN
 5248
 5249  REM BUILD
 5250 GOSUB 6100
 5251 RETURN
 5298
 5299  REM BURN
 5300 PROCh(n1%)
 5301 IF h% = 0 THEN RETURN
 5310 IF n1% = 78 THEN GOSUB 6150 : RETURN
 5320 IF FNo(0,42)<>R% AND FNo(0,81)<>R% THEN
PRINT K$;"USE THE FIRE" : RETURN
 5330 IF FNb((2,8) THEN PROCit ELSE RETURN
 5345 PROCoin(2,4)
 5346 ? FNin = 0
 5347 RETURN
 5348
 5349  REM CUT
 5350 IF n2% = 0 OR ?(O%+((n2%-1)*3))<>99 THEN
PROCtry : RETURN
 5355 IF n2%<>48 THEN PRINT "NOT SHARP ENOUGH" :
RETURN
 5360 PROCch(77,35,6)
 5361 RETURN
 5398
 5399  REM DRINK
 5400 IF FNg THEN PRINT y$ : RETURN
 5420 IF FNb(1,64) THEN PROCitELSEPRINTS$ : RETURN
 5425 ? FNin = 0
 5426 PROCoin(1,128)
 5427 RETURN
 5448
 5449  REM DIG
 5450 IF n2%<>65 THEN PROCtry : RETURN
 5465 IF FNo(0,65)<>99 THEN PRINT x$;"A SPADE" :
RETURN
 5470 PRINT "THE SPADE BREAKS"
 5471 ? FNin = 65
 5472 RETURN
 5498
 5499  REM DOUSE
 5500 IF R% = 78 OR (R%>55 AND R%<65) THEN PRINT
"IT BURNS FIERCELY" ELSE PRINT "FIRST FIND A FIRE"
 5545 RETURN
 5548
 5549  REM EMPTY
 5550 IF FNg THEN PRINT y$ : RETURN
 5565 IF FNb(1,48) THEN PROCit : PROCoin(1,-32) :
```

Figure 9.8 *(continues)*

```
RETURN
 5570 IF FNb(1,32) THEN PRINT Q$ : RETURN
 5575 IF FNb(1,16) THEN PRINT l$;"EMPTY" : RETURN
 5580 RETURN
 5598
 5599   REM EAT
 5600 IF FNg THEN PRINT y$ : RETURN
 5620 IF FNb(1,128) THEN PROCit ELSE PRINT "YOU´LL
BE SICK" : RETURN
 5630 ? FNin = 0
 5631 PROCoin(1,64)
 5632 RETURN
 5648
 5649   REM FIGHT
 5650 GOSUB 4550
 5651 RETURN
 5698
 5699  REM FILE
 5700 PRINT J$;"NO FILING CABINET"
 5701 RETURN
 5748
 5749  REM FLY
 5750 PRINT p$
 5751 RETURN
 5798
 5799  REM FAN
 5800 IF FNo(0,6)<>99 THEN PRINT x$;"A FAN" :
RETURN
 5820 IF R% = 78 OR (R%>56 AND R%<65) THEN PRINT
"THE BLAZE BURNS THE FAN" : ?(O%+15) = 0
 5840 RETURN
 5848
 5849  REM FLOAT
 5850 v% = 39
 5851 GOSUB 5950
 5852 RETURN
 5898
 5899  REM GIVE
 5900 IF FNg THEN PRINT y$ : RETURN
 5905  IF nl% = 57 AND R% = 17 AND FNo(0,57) = 99
AND FNo(0,48) = R% THEN PRINT V$;"HIS KNIFE" : nl%
= 48 : ? FNin = 99 : ?(O%+168) = 0 : RETURN
 5908  IF nl% = 15 AND R% = 81 AND FNo(0,15) = 99
THEN PRINT "S";V$;"A GOLDEN JAR" : ? FNin = 0 :
?(O%+27) = 99 : RETURN
 5910 IF nl% = 40 AND R% = 49 AND FNo(0,40) = 99
THEN PRINT "S";V$;"A RUBY" : ? FNin = 0 : nl% = 7
: ? FNin = 99 : RETURN
 5911 n3% = nl%
 5912 PROCch(60,73,4)
 5913 PROCch(73,79,3)
 5914 PROCch(13,79,3)
 5915 PROCch(67,80,1)
 5916 PROCch(44,80,1)
 5917 PROCch(1,7,5)
 5918 PROCch(7,73,4)
 5920  PROCch(20,31,4):PROCch(61,67,5)
 5921 nl% = n3%
 5922 ? FNin = R%
 5923 RETURN
 5948
 5949  REM LAUNCH
 5950 PROCch(18,21,4)
 6000 RETURN
 6048
 6049  REM LURE
 6050 IF R% = 8 AND ?(O%+21) = 99 THEN PRINT "THE
WASPS EAT THE HONEY AND DIE" : L% = 2 : qu = TRUE
: ELSE PRINT "THEY DO NOT COME"
 6090 RETURN
 6098
 6099  REM MAKE
 6100 IF n2% = 0 AND nl%<>18 THEN PRINT Z$; FNvb;"
A LASSOO" : nl% = 22 : PROCoin(2,2) : ?(O%+63) =
14 : ?(O%+144) = 99 : RETURN
 6130 IF FNo(0,22) = 99 AND FNo(0,50) = 99 THEN
PRINT Z$; FNvb;" A RAFT" : nl% = 22 : ? FNin = 22
: ?(O%+27) = 50 : ?(O%+51) = 99 : ?(O%+147) = 0 :
RETURN
 6140 PRINT x$;"THE EQUIPMENT"
 6141 RETURN
 6148
 6149  REM MELT
 6150 IF n2% = 0 THEN PROCtry : RETURN
 6160 IF R% = 12 AND (nl% = 37 OR nl% = 78) AND
(n2% = 74 OR n2% = 42 OR n2% = 81) AND FNo(0,74) =
99 THEN v% = 43 : PROCch(nl%,12,2)
 6190 RETURN
 6198
 6199  REM MOUNT
 6200 PROCch(36,49,4)
 6201 PROCch(56,74,6)
 6202 IF R% = 8 THEN L% = 1
 6225 PROCh(nl%)
 6226 IF h% = 0 THEN RETURN
 6230 IF R%<>49 AND R%<>74 AND R%<>8 THEN PRINT
"NO ROOM TO FLY"
 6247  RETURN
 6248
 6249  REM SWIM
 6250 IF R%<25 THEN PRINT J$;"NOTHING TO SWIM IN"
: RETURN
 6270 IF nl%<>14 AND nl%<>19 AND nl%<>5 AND
nl%<>23 THEN PRINT H$ : RETURN
 6280 v% = 7
 6281 GOSUB 4350
 6282 RETURN
 6298
```

Figure 9.8 *(continues)*

```
 6299  REM SMASH
 6300 GOSUB 4100
 6301 RETURN
 6348
 6349  REM SMEAR
 6350 PROCh(nl%)
 6351 IF nl%<>54 THEN PRINT p$ : RETURN
 6365 GOSUB 5000
 6366 PRINT R$;"NOW COVERED IN MUD"
 6367 PROCoin(2,2)
 6368 RETURN
 6398
 6399  REM SKATE
 6400 IF R% = 46 AND (( FNo(1,62) AND 4) = 4) THEN
st = 1 : PROCch(nl%,46,1)
 6450 RETURN
 6498
 6499  REM THROW
 6500 PROCh(nl%)
 6501 IF h% = 0 THEN RETURN
 6520  IF R% = 13 AND nl% = 49 AND ?(O%+63) = 99
THEN PRINT I$;"CAUGHT ON A ROOT" : PROCoin(2,2) :
?(H+297) = 1 : ?(O%+63) = 14 : ?(O%+144) = 13 :
RETURN
 6525 IF R% = 45 AND nl% = 38 OR (R% = 45 AND nl%
= 59) THEN PRINT "THE COFFIN SHATTERS" : ?(O%+99)
= 45 : ?(O%+111) = 0 : RETURN
 6540 GOSUB 4200
 6541 RETURN
 6548
 6549  REM TIE
 6550 GOSUB 6100
 6551 RETURN
 6598
 6599  REM WAVE
 6600 PROCh(nl%)
 6601 IF h% = 0 RETURN
 6610 IF nl% = 6 PRINT R$;"MAGICALLY WHISKED AWAY"
: R% = 58 : RETURN
 6620  IF nl%=58 THEN PROCch(58,60,5) : E% = 1 :
RETURN
 6640 PRINT "VERY PRETTY"
 6641 RETURN
 6898
 6899  REM VARIOUS PEEK AND POKE FUNCTIONS
 6905 DEF FNb(by,v) = FNo(by,nl%) AND v
 6910 DEF FNo(rby%,x) = ?(O%+rby%+((x-1)*3))
 6915 DEF FNg = FNo(0,nl%)<>99
 6920 DEF PROCoin(wby%,wv%)
 6921 ?( FNin+wby%) = ?( FNin+wby%)+wv%
 6922 ENDPROC
 6958
 6959  REM IS OBJECT HERE?
 6960 DEF PROCh(no)
 6961 h% = 0
 6962 IF FNo(0,no)<>99 AND FNo(0,no)<>R% AND
FNo(0, FNo(0,no)+1)<>R% AND FNo(0,
FNo(0,no)+1)<>99 THEN PRINT y$ : ENDPROC
 6970 h% = 1
 6971 ENDPROC
 6980 DEF PROCn
 6981 k = nl%
 6982 PROCdis
 6983 ENDPROC
 6990 DEF FNin = (O%+(nl%-1)*3)
 6998
 6999  REM ROUTINE TO SAVE GAME
 7000 DEF PROCsa
 7010 PRINT "INSERT BLANK CASSETTE"
 7025 ?&5F94 = R%
 7030 *SAVE "OPALDAT" 5000 5FFF
 7040 PRINT "FILE SAVED"
 7050 ENDPROC
 7198
 7199  REM INITIALISATION AND GAME LOADING
 7200 DEF PROCst
 7210 *LOAD"OPALDAT" 5000
 7220 IF ?&5F94<>0 THEN R% = ?&5F94 ELSE R% = 1
 7221 n% = 0
 7222 O% = &5000
 7223 H = &510F
 7224 ve = &5368
 7225 nn = &54D8
 7226 pn% = 0
 7227 X% = 0
 7228 oxy = 2:st = 0
 7230 d% = 0
 7231 L% = 0
 7232 E% = 0
 7235 Z$ = "YOU "
 7236 R$ = Z$+"ARE "
 7237 C$ = Z$+"SEE :"
 7238 J$ = "THERE IS "
 7239 P$ = J$+"AN EXIT "
 7240 U$ = Z$+"HAVE "
 7241 I$ = "IT IS "
 7242 Q$ = I$+"SHUT"
 7245 w$ = "WITH WHAT?"
 7246 S$ = "DON´T BE SILLY"
 7247 dir$ = "NSEWUDILQ"
 7248 p$ = "IMPOSSIBLE"
 7249 H$ = "WHICH WAY?"
 7250 K$ = Z$+"CAN´T "
 7251 x$ = Z$+"DO NOT HAVE "
 7252 y$ = x$+"IT."
 7253 l$ = I$+"ALREADY "
 7254 N$ = " NOTHING INTERESTING"
 7255 V$ = "HE GIVES "+Z$
```

Figure 9.8 *(continues)*

```
 7270 ns$ = "DOWN UP   WEST EAST SOUTHNORTH"
 7280 qu = FALSE
 7281 conj = 0
 7290 ENDPROC
 8995  REM *******  DATA *******
 8996
 8997
 8998  REM ROOM DESCRIPTIONS
 8999
 9111 DATA "4a78+W7<7A3K7N"
 9112 DATA
",X7U3K4a4B+W7[7a8´6)$M3Z+W.,8-+W82/4",""
 9121 DATA "4a4H14871,5M*6+W8<3K8B3$5M",""
 9131 DATA "+W828H,X7U3K4a4B8Q6)8X6%",""
 9141 DATA "4a4H8£6)3b+W9)8´6)9.",""
 9151 DATA "4a4H*6&Z4*949:140Gl,5M",""
 9161 DATA "4a4H&>0G9?3K+Wl,"
 9162 DATA "4a78*D;£4/4a",""
 9171 DATA ",X0G9H6.3K+W90631,4a78&Z+L9T6)&>4a",""
 9181 DATA "+W9\lD9b:´5M453K0G9.:,",""
 9191 DATA "4a4H:25C0.:;6.3Z82:@6.3Kl\*,5M",""
 9201 DATA "4a4H,X+W7[2L3Kl$.,5M",""
 9211 DATA "4a4H,XSJ0G:L:Q4a78&Z#V:[",""
 9221 DATA "4Y4B+W2<3K-40$3_0G+$",""
 9231 DATA "+D:a5M+W;%,L3Kl\0$",""
 9241 DATA "4a4H&>0G9?3K+W,L3Kl\0$",""
 9251 DATA "+W;*682\6<,X+W;%;/;55M4a"
 9252 DATA "*6;>:G4B+W;E;L3K+W;S,X+W;W;\6)+W&Y",""
 9261 DATA "4a#@+W<+:Q<4/D5M4H<<6)<?<G<N5M"
 9262 DATA "5(4H2=(&5C3Z<V<\",""
 9271 DATA "4a4H,X0G%#3K0G,D"
 9272 DATA "0H4B<a6)+W=%",""
 9281 DATA "0G=.4B=55C3Z=<5M"
 9282 DATA
"874H(&5C3Z<V<\=C5M%>=I3K=P49=V6)3K=\24",""
 9291 DATA "4a4H,X+W/<5a=£6)",""
 9301 DATA "4Z4B+W34-D",""
 9311 DATA
"3b0G,42L>+5M+W3<>05C>60.0G/<","0G2,4B>;",""
 9321 DATA "4a4H,X0G%#3K0G2$>@",""
 9331 DATA "4Z4B+W>G5\>K3K>P5M3K<V/<",""
 9341 DATA "0G=.´´5M6=3_>Z5M3K>_6)/<",""
 9351 DATA "4Z4B+W3<?%6)*<3_+$"
 9352 DATA "0G2,4B?*6)?0",""
 9361 DATA "4a4H?45Cl%5P6%?:,X+W=\#G5C3Z0$5M",""
 9371 DATA "0G*\?C6.?I0G?Q3K+W?V5C?\",""
 9381 DATA "4a4H@&5C*6+W@-0$",""
 9391 DATA "4a@4+W9:+W5G0G0$@96)-$142,",""
 9401 DATA
"4a4H,X0G@>@D@J:Q=\245M49@Q5M@V,X@[A#5M3KA(",""
 9411 DATA "0G@JA;5M,X6A+W0$6E"
 9412 DATA "+W@>3Z+W@D63AF:a5MAK6)*60G0$5M",""
 9421 DATA ""
 9431 DATA "4a4H,X0GAPAZ3K+W?V5C?\",""
 9441 DATA "+WB´=(5C,X=\2449=P:a5M:,",""
 9451 DATA "4a#@+W7N3K;\6)24",""
 9461 DATA ",X+WB<3KBBA(4a78+WBGBP6)BV2," ,""
 9471 DATA "4a4H,X0GB]C#",""
 9481 DATA "4a4H,X+WC(3KB]",""
 9491 DATA "4Z4B+WC/C9C@5C49CE5C",""
 9501 DATA "4a4HCK,X+WCR1L3K2,CZ6)4a+4C_5M",""
 9521 DATA "4a4H*6&Z)TD3&£6)*<",""
 9531 DATA "0GD3?C6.,X6A+WD8&>0GD>3KDC,L3K)T",""
 9541 DATA "4a4H,XSJ+WDM3K)T",""
 9551 DATA "4a4H,X+WEZ:L",""
 9561 DATA "4Z4B+WDVD]l\49E:",""
 9571 DATA ",XDbE´E/E55M3KE:)TEF4*4a",""
 9581 DATA ",X0G2<3KDb:LEKEO5M+W0LEU"
 9582 DATA ",X+WEZE£0HF%5Ml\-\%S5MF)F/",""
 9591 DATA "*6+WF73KF<FF5M0GFJFO0,-,"
 9592 DATA "#´FTFX5MF^6%G%6)5HG/G65M",""
 9601 DATA "4aG;3b+WGC3K?46)0L",""
 9611 DATA "0G7I3KDb:LEK:´5M%HGMGS",""
 9621 DATA "4a4H,X+WH,H3H83KIQ6))T",""
 9631 DATA "H=0G2<5M49HB3KDb:LEKHHHMHQ+,3K)T0\45"
 9632 DATA "&ZI(I-4BIl-$,X,-",""
 9641 DATA "4a78+WDCI>3K)T",""
 9651 DATA "4a4HID5C0.+W7<;£3K?*",""
 9661 DATA "4a4HIL6)0.0G3<3K2T",""
 9671 DATA ",XDbIQ6):LEKIVI\5M3KIbJ*H=0G736)"
 9672 DATA "A-J5,-J:6.0G=.$V-<6.CR6%",""
 9681 DATA "2TJ@5MH=0G7I3KDb:L",""
 9691 DATA "JKDb:LEKJR6.+W2<3KJZ",""
 9701 DATA "+WJ£3KK%2TK)5M3_#_6T4a",""
 9711 DATA "4a4H,X0GK/3K0G9\"
 9712 DATA "0G9\EO5MK76)0G-\5M",""
 9721 DATA "JKDb7<:L>+5M+W=\3KK<I(",""
 9731 DATA "4aKC,X6A0G.4#G5C3ZKK5M3KIb",""
 9741 DATA "4a4HKP+W5G0.+WKVlL3D",""
 9751 DATA "4a4H*6+W5HL29:L96%#N6)K8626%+4C_5M"
 9752 DATA "+WLF3KJZLK5MH=+WLP,X0G9:",""
 9761 DATA "4a4H*60GlWLY",""
 9771 DATA "MC+WMJlWMS5M49O]5M&RMa4a",""
 9781 DATA "4a4HEO6)*6+WN´6);£",""
 9791 DATA "Db4B0G2L3KND",""
 9801 DATA "4a4H,X+WNQ3KNVN]",""
 9811 DATA
"@&5M3K;£49O94HlL5C0.O=CR3D5M4E5(OD4a:5DbOQ",""
 9821 DATA "4a4H*60G7*6OOV",""
 9831 DATA "&ZO]3Z+WP#6)P)FF5M*6+W9b3K)D5M",""
 9841 DATA "IVLP5M,XDb*D;£*F6JP.5M3K0G7N%H6J"
 9842 DATA "+W7*3KlD5M4BP36)4E0GLP5MP;P?",""
 9851 DATA "4ZEO5M0GFY9ClL3D",""
 9861 DATA "4Z4B0GE´3KPJ5M",""
 9871 DATA "Db4B0GPQC(",""
 9872 DATA ",X0GGYEO5M+Wl,3K0LQ.5M",""
 9881 DATA "Q46)H=;£3_;£4B+WK\3Kl\-\",""
 9891 DATA
```

Figure 9.8 (*continues*)

```
"4aQ9Q=6%784PDbE´4UQB5M*60G736)49QK5M*60GHB",""
 9901 DATA "DbE´4B+WQP3KQZ>6"
 9902 DATA "*6+W543Kl\R8FF5M+WRE3ZRKRR5M",""
 9911 DATA "0GHB3KDbRWE´4B+WR^3KB]49S%"
 9912 DATA
"4/0G9C5>S.3K0GS46)HB&£5M+WS9503Kl\S?:Q+WSD=(5C,X*
D=I4B#_535;",""
11998
11999  REM SPECIAL MESSAGES
12000
12321 DATA "4a#20G+<"
12544 DATA "4aH33b"
12545 DATA "4a#2,X3_F)JZ"
12547 DATA "),4a#N"
12553 DATA "4aH33b"
12582 DATA "4aH33b"
12714 DATA "0G*L-T@94a"
13526 DATA "<a6)LP5M,X0GJZ"
13872 DATA "4a%Z+WLP49KC3b"
14777 DATA "4a%ZLP5M,X0G24"
15801 DATA "0G+L%)5M0G(T49%)5M4a+40Gl,"
15807 DATA "0GO]%)5M4a3b"
15813 DATA "0GRE%)5M$V"
15820 DATA "0G@>$F5M$V5M/4"
15861 DATA "5(&=0G0449$80G;£"
15867 DATA "4Y4B+WBG3K>6R8"
15873 DATA "+W*4l\IQR85M0G6="
15918 DATA "4a4HKP453_=\"
16378 DATA "4aH33b+WLP̄"
16436 DATA "$VP35M3Z4a,X$V5MG6"
16456 DATA "4aP33b0G;£"
16814 DATA "4a(-JK"
17258 DATA "4Y4H0MLP5M,X0G2T"
19997
19998  REM OBJECT DESCRIPTIONS
19999
20000 DATA "&Z(T"
20001 DATA "+W(\"
20002 DATA "+W)$"
20005 DATA "+W&Y"
20006 DATA "+W0,)D"
20007 DATA "87)L"
20009 DATA "+W)\"
20010 DATA "+W*$"
20011 DATA "87*,5M"
20014 DATA "&Z*D.,"
20015 DATA "87*L-T"
20017 DATA "+W*\"
20019 DATA "87l\+,"
20021 DATA "+W+<"
20024 DATA "&Z+T/$"
20025 DATA "+W+\"
20026 DATA "+W,$"
20032 DATA "+W,T3K)T"
```

```
20033 DATA "+W,\"
20037 DATA "87-<6)0$5M"
19968 DATA "+W&K"
20042 DATA "87.$-T"
20045 DATA "+W.<"
20046 DATA "+W.D"
20047 DATA "+W.L"
20048 DATA "+W.T"
20049 DATA "87.\5M"
20051 DATA "+W/,"
20053 DATA "87/<"
20054 DATA "+W/D"
20055 DATA "+W/L5M"
20056 DATA "+W/T"
20057 DATA "+W/\"
20060 DATA "8704"
20061 DATA "87(-5M"
20062 DATA "+W0D"
20063 DATA "870L"
20064 DATA "+W0T"
20065 DATA "+W2D0\"
20066 DATA "871$.,5M"
20072 DATA "871T"
20073 DATA "+W1\-\"
20075 DATA "872,"
20077 DATA "+W2<"
20082 DATA "873$5M"
```

Figure 9.8 The Opal Lily

Adventures and artificial intelligence

'This book is about the intelligence of computers, and about the intelligence of people who play computer games. On the one hand, it aims to show some of the basics of artificial intelligence and how it can be used in games. On the other it shows how you can go about writing games that require some intelligence to play. Of course, the two things go together.'

This is how Noel Williams introduces his book. Writing a good adventure game is by no means a trivial job, and using some of the most recent techniques, it can form an ideal introduction to artificial intelligence — the design of a program that makes the computer appear to react intelligently, learning from past mistakes. The book assumes a knowledge of BBC BASIC, implemented on the Electron or on the BBC Micro. All aspects of the design and writing of an adventure game are covered, including techniques of data compaction, sentence processing, and semantic databases.

As a final illustration of the techniques described, there is a listing of a full-sized eighty-one room adventure, The Opal Lily, occupying practically all of the Electron RAM. Other illustrations — with full listings — are Mernar Keep (a simulation/war game), and Dilemma, an original computer board game.

For the serious Electron or BBC Micro programmer, games writer, or student of artificial intelligence, this book is a 'must'!

McGRAW-HILL Book Company (UK) Limited
MAIDENHEAD · BERKSHIRE · ENGLAND

07 084749 5