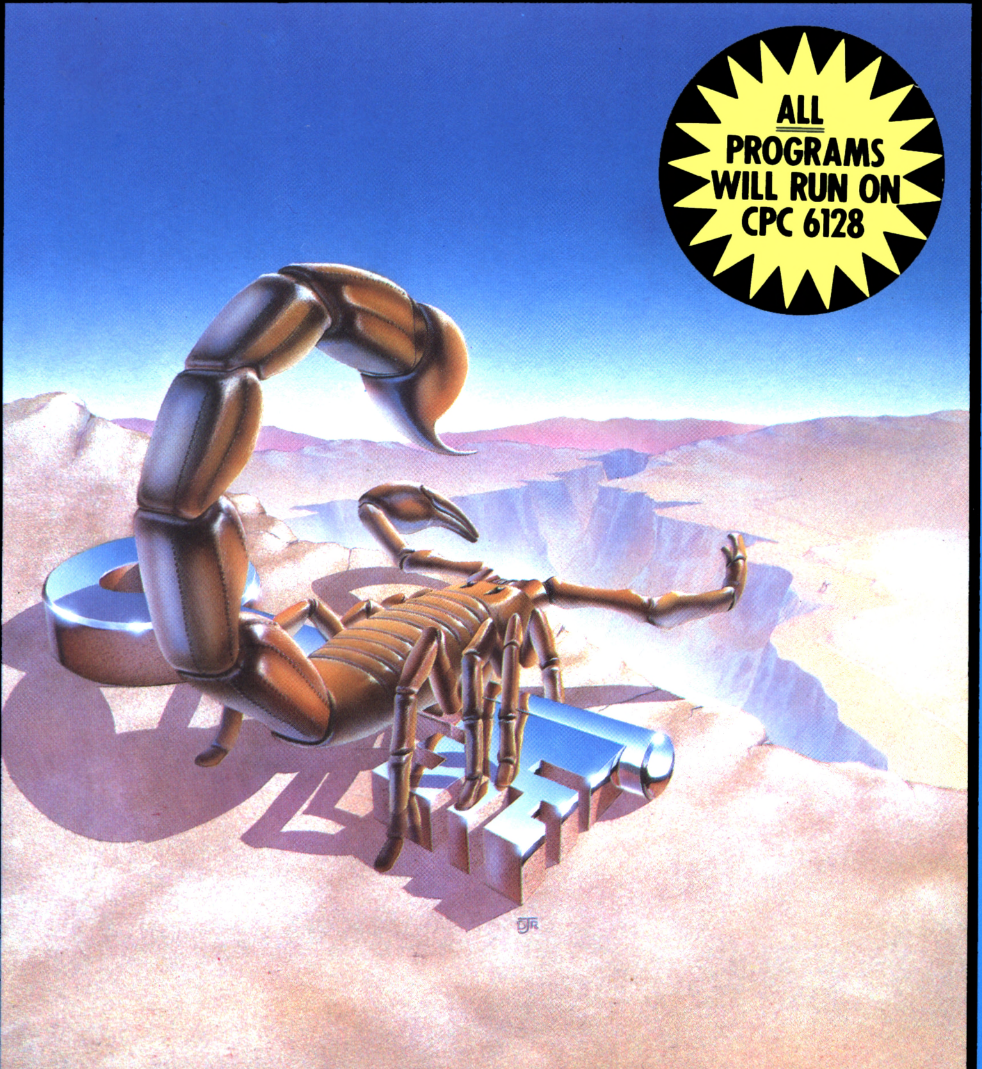
ARGUS BOOKS

ADVENTURE PROGRAMMING ON THE AMSTRAD CPC464 & 664

STEVE LUCAS

ALL PROGRAMS WILL RUN ON CPC 6128



Adventure programming on the AMSTRAD CPC464 & 664

Adventure Programming on the AMSTRAD CPC464 & 664

Steve Lucas

ARGUS BOOKS

Argus Books Limited
1 Golden Square
London W1R 3AB.

Series Advisory Editor: Dave Carlos

ISBN 0 85242 856 1

Phototypesetting by
Contact Typesetting Limited,
14 Stoneleigh Park Road, Ewell, Epsom, Surrey.
Printed and bound by Whitstable Litho.

Preface

After owning a computer for some time, most people reach a point where the novelty of 'zap em up' arcade games starts to wear thin. At this stage, many owners turn to adventure games as a source of more lasting enjoyment. For many people, the challenge of exploring a strange and mysterious land inhabited by even stranger creatures provides them with the escape from everyday existence that is sorely needed. For less than the cost of the airfare to an exotic island in the Pacific, you can be transported, in spirit at least, to places beyond your wildest dreams. One day you may be entering the jungles of the Amazon, and the next, flying in a spaceship to Mars. The challenge of exploring a land where your every move may be your last is one that few can resist.

Over the last few years, we have seen massive improvements in commercial adventure games, some of which have been due to radical changes in philosophy, whilst others have come as a result of constant refinement of adventure techniques. To new owners of an Amstrad microcomputer, the prospect of writing an adventure program of their own may seem to be beyond the bounds of feasibility, but in fact the programming skills needed to create a text only adventure are not unduly difficult and even a graphical adventure is not beyond the bounds of possibility.

Writing an adventure game is very similar to writing a novel. Everybody can write a few unrelated sentences, but the novelist's skill comes from stringing sentences together in such a way as to create a tale combining imagination, flair and ingenuity. The version of BASIC used in the Amstrad computers is known as Locomotive BASIC and it is one of the most sophisticated versions of the language on the market. It contains so many features to make life easier for the programmer that even full high resolution adventures are within the capabilities of the average programmer. With such a powerful tool at our side, the technical skill of the programmer is no longer the limiting factor in the process of designing an adventure which can be enjoyed by all. In this book you will be shown how an adventure game can be written by combining a number of standard routines together and thus the skill of designing an excellent game comes from creating a good plot rather than from the technical skill of programming.

Many of the routines found within the pages of this book can be taken and used within your own programs, although you may need to adapt them to suit the theme of your own game. There are many different ways of writing an adventure game and I have attempted to introduce as many alternative techniques as possible to show how they can be used to effect. To illustrate how these routines can be combined to produce a large adventure, I have included three full adventures and take you step by step through all the stages involved in their development.

Naturally, you will need to have a reasonable knowledge of BASIC programming before you will feel really confident to tackle you own adventure game from scratch, but even if you are an absolute beginner, without any knowledge of BASIC, you should find something of interest within these pages. Computer novices eager to make a start on their own program will find that the third program listing in this book was written just for them. The game is called *A Journey Through Space* and loads into your computer in two parts. The first part of this program is a standard adventure containing all the code necessary to control the game, whilst the second part is a data file created by a separate program which is listed at the end of this book. The data file contains details of all the locations, objects and words recognised in the game, and the program used to create it contains facilities for altering the file. When run, this program displays a description of each location and every object found within *A Journey Through Space* and asks you to type in any changes you would like to make. If you were to change all the descriptions in the file, you would, in effect, have created a completely new adventure without actually programming it. Once you are satisfied that the data typed in is all right, the program will save the file onto tape or disc so that it can be loaded in as the second part of the main game. In this way you can write a game of your own without any of the fuss.

Over the last couple of years, adventure games have improved beyond all recognition. Not only do many adventures now contain full high resolution graphics and exciting sound effects, but many now also have the ability to analyse full English sentences. No longer are we limited to giving the computer simple instructions such as GET LAMP, but can instead type more complex commands such as TAKE THE GREEN LAMP FROM THE TOP SHELF AND LIGHT IT WITH THE MATCH FOUND ON THE TABLE. In the final stages of this book, you will find a few clues to point you in the right direction for writing such routines in BASIC, although, in practice, you will probably find insufficient room. 64K Amstrad computers have about 43K of RAM free for BASIC programs if not fitted with discs and, although this is far more generous than many other machines, you will usually be forced to write your game in machine code if you want to include plenty of puzzles, high resolution graphics, sound and full sentence decoding in one program. Although assembly language and machine code programming are beyond the scope of this book, adventure games written in BASIC can be challenging to play and despite the limitation of the language, the response time should still be fast enough for even the most discerning player.

Contents

Programs

1 The Wizard's Quest Chapters 1-7

This is a traditional text only adventure game. An ideal introduction to adventure programming, illustrating many of the techniques of setting puzzles for the player to solve.

2 Snow White Chapters 9-12

A game for children based on the traditional fairy tale, featuring full high resolution pictures for each location in the game.

3 A Journey Through Space Chapters 13-14

In this game, the data is loaded from tape or disc to allow the player to modify the game without all the effort of starting from scratch.

4. Filer Chapter 15

This program is used to save a data file containing the starting position for *A Journey through Space.* When it is run, you will be asked whether you want to modify the game and if you answer 'yes', the program will allow you to change any (or all) of the locations and objects in the game.

Why Amstrad?

The Amstrad CPC 464 and the CPC 664 contain a very powerful and sophisticated version of the BASIC language which has been specially written for the machine by Locomotive Software. One of the most exciting features of these machines to the adventure programmer is the massive amount of memory free for use in BASIC. In many other home micros which claim to be 64K machines, very little of this RAM is available for use in BASIC. Only a couple of years ago, most computers had only 16K of RAM, or even less, free for BASIC, yet some very sophisticated adventures were written for 16K TRS80's and Commodore Pets and even the 16K ZX81 was capable of running some fairly exciting adventures. Since then, however, the average computer owner has come to expect games which include full high resolution graphics, excellent sound effects and full sentence decoding. Adding these features to an adventure does tend to demand a massive amount of RAM and unless the program is written in machine code, it is unlikely that you will be able to fit them all into a single game.

Where Locomotive BASIC scores over many of its rivals is in having a wide range of commands available to control graphics and sound. Only one feature is sadly lacking in the BASIC commands of the CPC 464 and that is a FILL commmand. Drawing a shape and filling it in will take a little more programming on a CPC 464 micro than on some of its rivals. The later CPC 664 has remedied this shortcoming. Nevertheless, the CPC series of micros leaves over 43K of RAM in which you should be able to write a massive adventure game and still find room for graphics and/or sound! If you compare the Amstrad micros with other machines, you will see just how powerful they really are. The Commodore 64, for example, contains the same amount of user RAM, but only 38K or so of it is available for the user's BASIC program. To make life even more difficult, the commands for manipulating the excellent graphics are virtually nonexistent in BASIC, and all the effects have to be produced by POKEing data into memory. Yet another difficulty with this machine for the adventure programmer is that it will not accept more than about 80 characters in any one line. The BBC Micro and its baby brother, the Electron, on the other hand have some very powerful graphics and sound commands available, but leave only a fraction of the memory available on the Amstrad. Even the ever popular Sinclair ZX Spectrum fails to offer all the features of the Amstrad computers. At the moment, the Amstrad computers are still relatively new and although a few commercial adventures are available, many of these are merely conversions of programs from other machines. Not that a game should be rejected on these grounds alone, for in fact the programs I've seen so far, including the excellent games from Level Nine Computing and the superb version of the Hobbit from Melbourne House, really do give Amstrad owners something to get their teeth into. In addition to these conversions, there are also a number of excellent adventures which have been specifically written to make use of the features of the Amstrad computers. The series of adventures featuring Arnold Blackwood by Nemesis Software and the graphics adventures from Interceptor Software are just two examples. What is disappointing to the adventure game enthusiast is the small number of such games, and what better way of rectifying this lack of software than by writing your own. Maybe you can even produce the next masterpiece?

Although the graphics and sound commands allow excellent refinements to be made to games, it is in the string handling commands that Locomotive BASIC really comes into its own for writing adventure games. As you would expect, the Amstrad computers contain all the usual string handling commands such as LEFT$, MID$ and RIGHT$, which allow the programmer to manipulate the text part of the game. In addition, however, there are a number of other facilities which are not so widely available in home computers. The INSTR command is one of the most useful commands and this makes writing routines to analyse the player's instructions much easier.

Despite the massive amount of RAM free for the programmer to use, the arrays used in adventure games will rapidly eat up memory space and the programmer will probably still want to save every byte of memory, so as to pack as many features into the game as possible. There are many ways of saving memory space and the short list below should point you in the right direction.

1 Remove all spaces between key words. This is a very effective way of saving RAM, but does make it more difficult for a computer novice to type in. Consider the two examples below.

```
10 IF(P%=88ANDR=3)OR(P%=76ANDR=2)ORR=84THENGO
SUB2000:GOSUB2010:PRINT"O.K."R=3:S%(89,4)=27: RETURN
```

```
10 IF (P%=88 AND R=3) OR (P%=76 AND R=2) OR R=84
THEN GOSUB 2000: GOSUB 2010: PRINT"O.K.":R=3:
S%(89,4)=27:RETURN
```

If you were to try typing in the first listing, the Amstrad would produce a SYNTAX error, because there must be either a space or some other delimiter between keywords. The second listing is easier to read, but there are some additional spaces which can be removed. See if you can find out which ones are *not* absolutely necessary.
2 Remove all REM statements. These are totally unnecessary to program operation and any program will work equally well if they are left out. In this book, however, REMs are used extensively because they do help to explain what each section of the program does.
3 Use integer variables wherever possible. They are far more efficient in their use of memory and are little trouble to use. In any adventure game, the numeric variables are going to refer to the number of a location or object and, as you are never going to have an object number 2.4 or a location number 17.9 you may as well use an integer variable. All that is necessary is to put a % sign after the name of the variable. Thus I have used P% rather that P for the number of the location in all the adventure listings in this book.
4 Long variable names do make listings easier to follow than single letter variables, but they also demand rather more memory. Rather than use variables like map%(place%,direction%), I have used S%(X,Y). This also saves typing time!
5 Use the zero element of arrays! Many programmers, myself included, tend to ignore the zero element of an array. Whenever an array is dimensioned, the computer will leave room for the zero element and if it isn't used, you are wasting RAM. Suppose, for example, that you DIMension the array A$ at the start of the program with a line such as:-

```
10 DIM A$ (40)
```

There would be 41 elements available for use from A$(0) to A$(40). Using the elements from 1 to 40 does, however, make for easier programming!
6 One final feature which is unique to the Amstrad is the facility for converting a string to lower or upper case with the commands LOWER$ and UPPER$. When the player types in the instructions for the next move, he may well use either capital or small letters and unless you use these commands to convert the sentence into lower or upper case, you will find yourself writing a far longer section of code than is necessary.

Introduction

Where have adventures come from?

Although the very first adventure game was written as far back as 1976, it wasn't until the arrival of the cheap home micro that adventures started to become really popular. The very first adventure was written on a large mainframe computer at Stanford University in the USA by two computer enthusiasts. Don Woods and William Crowther's original game was written in FORTRAN and, unlike BASIC, this language does not contain facilities for handling words. Despite the limitations of the system, these two experts managed to create a game which has stood the test of time and even today is a firm favourite with computer buffs. Using FORTRAN as a language meant that the data for the game had to be stored on disc and over 250K of memory was needed to play the game. It is little wonder, therefore, that before Apple, Tandy and Commodore started to produce microcomputers in the late 1970s few people had even heard of an adventure, let alone played one. Only those lucky enough to have access to large mainframe computers in colleges, universities and large companies were able to experience the delights of killing snakes and catching little birds.

When production of microcomputers started in the late 1970's few people thought that it would be possible to write an adventure game which would run on such a machine, after all it required over 250K of memory and large disc drives to run the original game, often referred to as *Colossal Caves*. Scott Adams, a young American, was the first person to realise that it was feasible to write an adventure game for the microcomputer and went on to produce a now famous game called *Adventureland* for the Tandy TRS80. It was this game that really convinced large numbers of computer owners that adventure games were fun, and Scott Adams has since gone on to write a whole series of adventures. His company, Adventure International, has written and adapted these games to run on a wide range of microcomputers and has started to add high resolution graphics to many of them. Unfortunately, at the time of writing, they are not available for Amstrad micros, although I'm sure that if enough people demand versions, we'll see them in the shops very soon. I, for one, look forward to the day when I can play *Pirate's Cove, Strange Odyssey* and *Preppie* on my Amstrad. Many software houses have attempted to convert the original *Colossal Caves* to run on microcomputers, and Amstrad owners are now able to buy at least two versions. My own favourite is produced by Level Nine Computing, a British software house famous for their superb adventure games, who have managed to cram over 70 extra locations and a new 'end game' into their verson of the original game. In order to compress such an enormous amount of data into an Amstrad microcomputer, they have written the program in a specially created adventure language called 'A-Code'. If you are new to adventuring, there can be no better introduction than this exceptionally well produced game.

What is an adventure?

If you've never played an adventure game before, you're probably wondering what I'm talking about! Just in case you are puzzled, I'll try to give you a brief explanation. An adventure game is rather like a story in which you play the leading role. As you type instructions on the keyboard of your computer, the tale will unfold before you. A good decision will lead you further into the game, where you will encounter all manner of puzzles and problems to solve. The computer transports you from the comfort of your armchair into a new, and often hostile, land where many strange creatures are to be found.

An adventure game is, in many ways, like a book. It should have a good story line or plot, be well written and, most importantly, be enjoyable. Unlike a book, however, the sequence of events will be different every time it is played. The computer will act as your eyes and ears, telling you of any dangers you are likely to face and even giving you help when you need it. There is no substitute, however, for playing a game and no matter how much I try to explain what an adventure is, you will only really find out for yourself by playing one.

Trying to explain what an adventure game is to someone who has never played a game before, is made even more difficult by the fact that adventures have changed over the last few years. There are now so many different types of games available that an adventure can have many different meanings. All the early programs were text only games in which the locations, creatures and objects were described to the player in great detail using words alone. There have been many improvements in this type of adventure over the last couple of years. Since Melbourne House released probably the most famous adventure game of all, *The Hobbit,* most new games have tried to include the features found within this game. Few have succeeded in finding the formula which made *The Hobbit* such a popular program. Whether it is the fact that it's based upon a famous novel, or whether it's the sheer quality of graphics, there is no denying the fact that this game is still one of the best adventures around and, fortunately, is now available for the Amstrad. There can be little doubt that good high resolution pictures can transform a very good game into an excellent adventure, although no amount of fancy graphics can convert a poor game, with little plot, into even an average program. Adventure games now fall into many different categories, ranging from the traditional text only game, based on the original Crowther Wood's program, through the more modern graphical adventure with full sentence decoding, to the role playing adventure.

Many people would argue that a role playing game is not a true adventure at all because the player is limited by the nature of a character given to him at the start of the game rather than by his own cunning and ingenuity. Often these role playing games are based upon *Dungeons and Dragons* and involve the player fighting other creatures which he or she comes across in the game. A true adventure, on the other hand, is much more of a 'mind game', involving puzzle solving rather than chance. Personally, I have yet to come across a really good implementation of a *Dungeons and Dragons* game on a microcomputer and, for that reason, I have avoided trying to develop such a game myself.

Many adventure game enthusiasts are equally critical of the modern graphics adventure, claiming that the mind is capable of conjuring up far better pictures than any computer VDU. Whilst I would agree that some of the text adventure games are superb, there are also a growing number of very good graphical adventures. In this book, I shall show you how to develop both text and graphics adventures.

Writing your own adventure games

Once you've decided to take the plunge and write an adventure of your very own, there are several decisions you'll have to make before you reach the point of sitting at the keyboard and typing the game in. The very first thing you will have to choose is whether you are going to write a text only or a graphics game. Despite those cynics who seem to despise anything other than the traditional game, graphics adventures are great fun to play and even more challenging to write. If you are fairly new to programming, however, I would suggest that you start off with a traditional text only game rather than throw yourself in at the deep end. After a few games, you will be only too eager to write a game with full graphics to illustrate each location.

One point worth bearing in mind before making the decision as to whether a game should contain graphics or not, is the vast amount of memory needed to include pictures. Even on a machine with very powerful graphics commands, like the Amstrad microcomputer, the space left when you have drawn the pictures for each location will limit the rest of the game to such an extent that you may only be able to fit a quarter of the number of locations and puzzles into your program. In many commercial games, like *The Hobbit,* the authors have realised the limitations and have made the decision to include graphics for only a few of the locations, the rest being treated as if it were a normal text game.

Whether you write a graphics or a text adventure, the most difficult part of the whole process lies in choosing a good plot, rather than in the actual coding of the game. No amount of fancy graphics or detailed text will improve a game with a poor plot. Early games tended to follow a very simple theme which involved moving around a strange world inhabited by dangerous creatures and gathering items of treasure to take back home to safety. The original adventure games are probably more popular today than they were a few years ago, which just goes to show that it is still possible to take a simple theme and transform it into a superb game. As adventurers practise their skills by playing more and more games, they are becoming ever more critical, and if you intend to stick to such a well worn plot, you will really have to pay great attention to the little details which can make all the difference between a poor adventure and an enjoyable game.

More modern adventures tend to have a much more tightly controlled plot, where scores are given for solving specific problems, rather than for simply finding treasures. In some games, the player can even lose scores for falling into traps and may solve the adventure without ever scoring 100%. Progress in these games may well follow a much more linear thread, where, once you have entered a new location, there is no way back again. In some games, time may also play a part. Imagine a game based on Cinderella, where the player must get back before the clock strikes midnight, or a game where the player presses the fire button on the space rocket and is unable to return to the plant to pick up the plutonium he needs for his later mission.

It is well worth while playing as many different adventure games as possible to give you a better idea of the sort of things which can be achieved on a micro, before attempting to plan your own game. This should give you a much better idea of what you want to achieve. The very best starting point for any adventure is to sit down with a pad and paper, a pencil and a rubber and write a short summary of the story for the game. In the first chapter, I shall show you how I took the basic plot for a game and transformed it into a map of 'Middle Earth' ready for conversion into the program itself.

Some ideas for adventures

Stuck for an idea? Then the list of suggestions below might just set you thinking and point you in the right direction to start your own games.

Lost horizons
Over the last few years, tales have started to reach you of a valley deep in the Amazon Basin, where it is rumoured that the secret of eternal life is to be found. Within the walls of a ruined city created by forces beyond the bounds of human knowledge, there is supposed to be a small temple where the secret scrolls are guarded over by the spirits. With dreams of unknown wealth and eternal life, many have set forth to find the valley, but none has yet returned! Will you be the first to find the city and return with the scrolls?

Journey through time
Two days ago you received a distress call from the people of the plant Ursa and, like all true Timelords, you could not let the people suffer in the hands of the evil Trell. Shortly after leaving your ship, however, two thieves entered the control room and stole the four crystals which control time travel. On your return to the ship, you find that the thieves have left just one small clue to the whereabouts of your only source of escape, a small piece of paper with strange writing on it. What does it mean? Can you recover the crystals and escape or will you be doomed to spend eternity on Ursa?

The vampire's curse
For many years the villagers of LLudnia in a remote area of Transylvania have been terrorised by the vampires in the dark and gloomy castle high on the hill overlooking the village. One day the villagers, led by the local priest, decided that they had had enough and set out at dawn, determined to rid the castle of its curse forever. Fritz, the local dentist, was wiser than most and realised that the traditional methods just didn't work. A stake through the heart and the crucifix had all been tried before and he knew that a new approach was needed. Armed only with his small bag of tools, he set out shortly after the others. In this game you must take on the role of the local dentist and try to return to the village with all the fangs before the vampires rise at dusk. Will you manage to succeed where others fail or will you too join the vampires in the castle?

Detective agency
You have recently set up your own private detective agency in downtown Bognor. Early yesterday morning, an old man entered your office telling you of a murder which had occurred in his house. The police had been called and seemed to think that all the evidence pointed towards one man ... your new client. Just ten minutes ago, you received a phone call from your client to tell you that he had been arrested and charged with the murder. You are convinced of his innocence and must try to find some new evidence. Can you find the clues needed to solve the murder, or will your client go to prison for a crime he didn't commit?

Castaway
It has been six days now since your ship sank in a violent storm. You are tired and close to death, drifting alone in a small lifeboat, when you see a small island in the distance. Quickly you grab the piece of driftwood and row ashore. What does the strange drum beat mean? Why do fish suddenly appear dead on the shore? Can you solve the mystery of the island?

If the ideas above don't really capture your imagination, why not take the plot from you favourite novel or short story. It's an ideal way of starting an adventure, although you may come into copyright problems if you try to sell a game written in this way. If you stick to traditional stories such as Robin Hood, or to fairy tales, you will not run into such problems and should be able to let your imagination run riot. For those who have no intention of marketing their games and who write for fun alone, there is absolutely no reason why the game shouldn't be based around any story or novel.

Do remember that to make a game enjoyable, the puzzles and problems set for the player should be relevant to the theme of the game. Thus games written about Sherlock Holmes can contain puzzles about violins, chemicals or murder, where as problems set in games based on James Bond should reflect high technology, secret agents and exotic locations. Several adventure games have appeared on the market recently where the puzzles are totally illogical, making the solution more a matter of luck than skill, and most of these programs have been doomed to commercial failure. Don't let yours fall into the same trap.

Most commercial adventure games are written in machine code rather that in BASIC, although a number of software houses have created their own languages specifically for writing adventure games. There are two main reasons why BASIC is often considered to be unsuitable for adventure games. Firstly, a game written in BASIC is often very slow, resulting in long response times. There is nothing worse than typing an instruction into your computer and having to wait thirty seconds or so for a response! If, however, you plan your routines very carefully, the response time can be almost as good as in machine code games. This is especially true of Amstrad machines, which have one of the fastest versions of BASIC around. The second, and probably most important, advantage of machine code over BASIC is that it is possible to cram far more puzzles and locations into a game.

Both machine code and adventure languages are beyond the scope of this book and although a game written in BASIC can't be as complex as a game written in machine code, it is still possible to write a game with over 200 locations and 50 objects to fit into the 43K of RAM available in Amstrad mircos if you are very careful in your approach. One final advantage of a machine code game is that it is much more difficult for the player to cheat and solve the game by listing it. This is a point I shall come back to later.

In many ways, adventure games are very similar to database programs. The computer must store information about the locations and the objects found in the game and one of the most useful methods is to store this information in DATA lines ready to be read into arrays. Before beginning to write your own adventure, you really do need to be familiar with the use of two dimensional arrays. In the process of creating an adventure game, you will certainly become a much more proficient and confident programmer! Don't be put off, however, if you don't feel very confident about the use of arrays and string handling. The third listing found in Chapter 13 should help you to write an adventure of your own without all the effort needed to write one from scratch.

In any large program, you will inevitably make many simple typing mistakes when you enter the program into your own computer. Rather than waiting until you have typed the whole program in and then trying to track down the errors made, it makes much more sense to type the program into your computer in short sections and test each one as it is entered. Each of the three programs in this book is split into short routines which can be checked out in this way before proceeding with the next one, and full instructions are given to help you debug the games.

Getting started 1

Writing an adventure game for your Amstrad computer will provide you with a challenge that is guaranteed to keep you out of mischief for many weeks, or even months. There are so many ideas to sort out that it will take several hours of preparation before you are ready even to begin programming the game. Some days you may feel that you are making rapid progress in developing your program, whilst on others you'll spend many hours trying to sort out a minor problem. When you do come across a problem which seems to be taking far too long to puzzle out, the best approach is to give up. After you've had a drink and rested your brain for an hour or two, you'll come back to the problem fully refreshed and ready to go! The time spent in developing a large adventure may be exciting, time consuming or even frustrating, but never dull or boring! In the process of writing your game, you are bound to learn a great deal about the operation of your computer, and this new found knowledge should encourage you to attempt ever more adventurous programs!

Although there are a few really good adventures for the Amstrad computer, there are many more which don't reach the same high standard and if you can find a really good plot, you are half way to writing a superb game. Finding a suitable story line is, in fact, the most difficult part of the whole process. We have already looked at some ideas for plots in the introduction, but after you've exhausted those ideas, what next?

Take a quick glance along the shelves of your local library and you will find thousands of books which fall into a few familiar categories: thrillers, science fiction, history, fantasy, westerns, detectives, horror, romance to name just a few. As I have already mentioned, adventure games are very similar to novels. Just as the author of a detective story can very often take a familiar theme and give it a new twist, so too can the adventure programmer. Very few adventures are based on a completely original idea, yet most of the really successful ones are written by programmers with a vivid imagination who have managed to take an old idea and present it in a completely new way. The quality of an adventure game is limited only by the imagination and skill of the programmer!

Adventure into education

The quality of most educational software is so poor that many teachers have rejected the computer in favour of more traditional forms of education. A few rather more enlightened teachers have realised that adventure games can offer a far more exciting education to children than many so called 'educational programs'. Unlike arcade games, adventures encourage logical thought and, if the game is really well planned, it can also be used to encourage children with map drawing, creative writing and problem solving. There are many recorded cases where the careful use of adventure games has helped to develop the potential of slow learning pupils, but as yet, adventures have not really been used in the normal classroom environment to any great extent. Most of the best adventures have been written without any regard to their educational content, and yet these very adventures can probably be considered to be some of the best educational software around. If only the programmer could set out with the intention of writing a good educational adventure, I'm sure we would see even more useful programs. Imagine, for example, a program based on historical facts: Guy Fawkes, Captain Cook or I.K. Brunel. You could even devise a program which required a knowledge of chemical formulae!

Don't be put off writing your game by those cynics who claim that there are too many adventures set in 'Middle Earth', or that adventures set aboard a deserted spacecraft are boring. If you can think of a completely new story line then so much the better, but even if it is based on a familiar theme, your program should still reflect your own personal blend of puzzles and problems and should be something to be proud of. Once you have got the basic framework of the game sorted out and got it running on your computer, then you can spend many happy hours at the keyboard refining the puzzles and, eventually, putting the finishing touches to your masterpiece so that it contains your own unique mixture of wit, humour and sophistication. At this stage however, all you really need to sort out is the basic plot and a few ideas about the nature of your game.

Programmers do tend to be an impatient breed. Eager to get their hands on the keyboard, they will often neglect the very important preliminary paper work. Time spent with a pencil, paper and a rubber at the planning stage is well spent because a program developed at the keyboard will inevitably lack structure and this in turn will make debugging a nightmare! Many programmers still look upon flowcharts as something to be avoided at all costs, but even a simple flowchart can help you to sort out your ideas and make program development much easier. Without one, you are likely to end up with a program totally lacking in structure and this in turn will make it far more difficult to follow when you do discover a mistake. In principle, adventure games are very simple in structure and Fig. 1.1 shows how the game may be broken down into simple, easy to develop, stages.

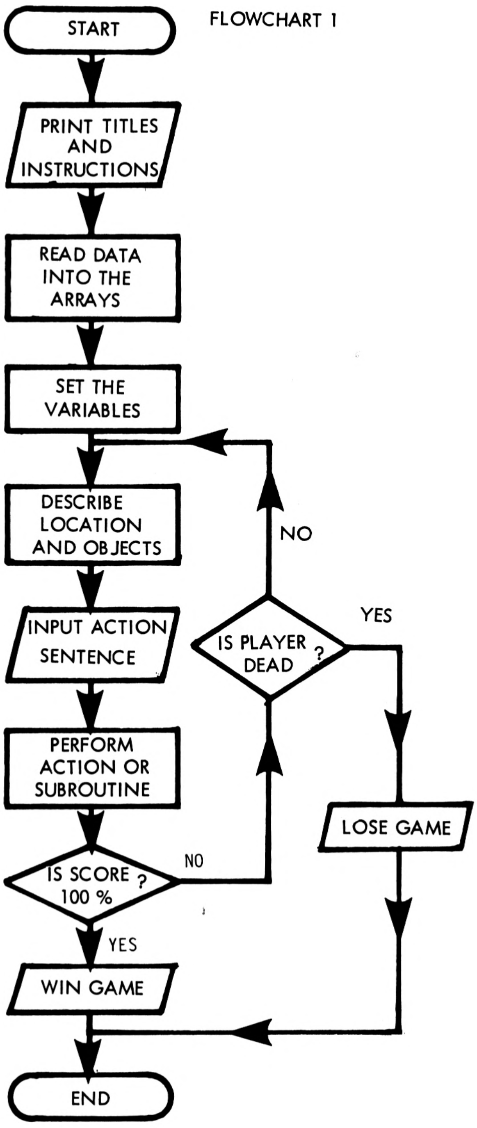


Fig. 1.1 Flowchart for typical adventure game.

The first thing you will notice if you compare this flowchart with any of the listings in this book is that I have not followed it to the letter, but have instead used it as a guide. A major difference between the flowchart and the listings is that I have not included instructions within any of the games. There are two reasons for this. Firstly, the programs in this book are written in BASIC and, in order to make the listings easy to understand, I have used plenty of REM statements and also left spaces between words wherever possible. BASIC is, unfortunately, very inefficient in its use of RAM and this can only make matters worse. Instructions within the program use valuable memory space which can be better utilised by adding extra locations to visit, objects to pick up and puzzles to solve. In the first game, for example, I had to decide whether to include the facility to save a game on tape or to incorporate instructions within the main game; in the end I decided that the save game routine was too important to leave out. Secondly, finding out what the game is all about is often an integral part of the overall puzzle. In practice, you can always write a short program containing the instructions which then loads and runs the main game, or, even more simply, write the instructions on paper.

The first program listing in this book, *The Wizard's Quest* is an example of one of the earliest types of adventure game, where the player must set out to explore the land and return with items of treasure. Each object of value gives a score of one when it is placed in the right location. This program, like most others, accepts only one or two word sentences and the player must type instructions such as GET LAMP or GO IN. Text adventures of this type really do need to be well planned if they are to be different from the rest. Descriptions of locations need to be very detailed, and the screen display should be as neat as possible. If you can include cryptic clues within the descriptions of locations, it does help to make the game more interesting. If this is the first time you've attempted to write an adventure game, I wouldn't try to be too ambitious. Success at writing a game with a fairly simple plot and just a few locations is far more rewarding than failure with a massive game.

Converting your ideas into a working program will require careful preparation, the first stage of which is to draw a map of the locations in the game. If you have decided to use a book as the basis of your game, this process should be fairly straightforward, although a plot of your own offers far more scope for originality. The map itself need not be very detailed, but before starting to draw it, it's worth considering some of the limitations of the Amstrad computers. Although there is over 43K of RAM free for BASIC programs, this will be very rapidly used up in an adventure game. The first listing in this book illustrates some of the compromises which have to be made. It contains 30 objects and 80 locations, all of which are described in great detail. Even with the save game feature, there is still about 20K of RAM free. I have included plenty of REM statements in the program to make it easier to follow, and if these are left out, you should be able to save even more memory, which can in turn be used to add extra features to the game.

Unlike the first game, the second program contains a full high resolution picture of each location, together with a few sound effects. Both sound and graphics routines tend to have a voracious appetite for memory and therefore a graphics game cannot incorporate as many locations as a pure text adventure. In *Snow White,* I have included just 24 rooms, which leaves about 20K of memory free for you to add a save game routine or extra puzzles.

The final listing shows how it is possible to load the data for the descriptions of locations and objects from tape or disc rather than keeping this information within DATA lines. This is a much more efficient method of writing adventure games but, unfortunately, programs written in this manner are far more difficult and time consuming to create. Not only do you need to write a second program to create the data file in the first place, but each time you make a slight typing error, you have to load the data file in again. From tape, this will take several minutes, although from disc things are much better. Pushed to the extreme, you should be able fit well over 250 locations into a game written in this manner, but if it's your first attempt, I'd be a little more cautious! The majority of adventure games are 'two dimensional', with most of the locations being on the same level. A few games, however, are truly three dimensional. In such a game, there will be several locations where the player can move up or down onto a new floor. Drawing a map for a three dimensional game tends to be a far more difficult task than for a two dimensional game, as can be seen in Fig.1.2.

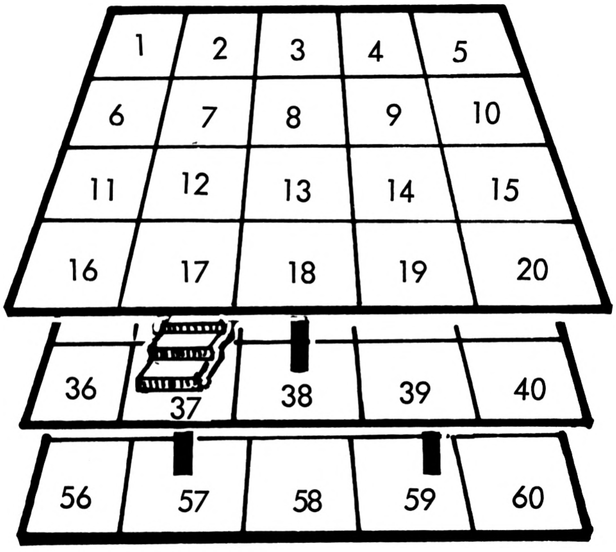


Fig. 1.2 Map for three dimensional game.

If your game contains fewer than ten locations where movement up or down is possible, there is no need to write a full three dimensional game. It is far easier to insert a short subroutine to deal with such movement than to incorporate it within the main program. This is an approach I have adopted in all the listings in this book. There is one other advantage over a full 3D game, where you would need to increase the dimension of the array used to hold the map and that is, of course, going to use more memory.

One other point worth bearing in mind when planning your map is the fact that allowing movement in directions other than prime compass points will use even more RAM. For this reason, I have not inclued the ability to move northeast or southeast in any of the games. You could, of course, try experimenting with these extra directions of movement. It shouldn't be difficult! Fig.1.3 shows one approach to map drawing for adventure writing. Each location is given a discrete number and I've used wiggly arrows to link locations which are reached by methods other than moving north, south, east or west. Thus to reach location 2 from location 3, you would have to swim. I've not included detailed descriptions of the locations on the map, as this would only make it more confusing. At this stage all that is needed is a few words to indicate the type of place. Detailed descriptions can be left to the programming stage. Fig 1.3 shows just 10 locations, but you can add as many rooms to your game as you like (within reason of course!).

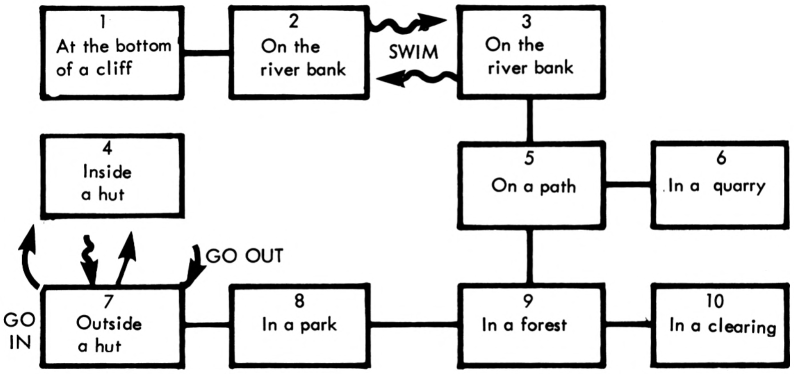


Fig. 1.3 Drawing a map for an adventure game.

Once you have drawn the full map for your own game, the descriptions of the locations and the directions in which movement is possible will need to be converted into a format suitable for inclusion in DATA lines. Before you rush off to the keyboard to make a start, you should give some consideration to the nature of the objects, creatures and other puzzles you are going to include in the game. I like to show these on the map right from the start and, in order to distinguish the objects from the locations, I try to use a different colour for locations, objects and puzzles.

Rather than continue to talk about a hypothetical game, I shall now refer specifically to the first game, *The Wizard's Quest*, so that you can see how I set about writing it. In so doing, I shall discuss the solution to the game, so if your prefer to solve it on your own, you should jump straight to the listing in Chapter 2.

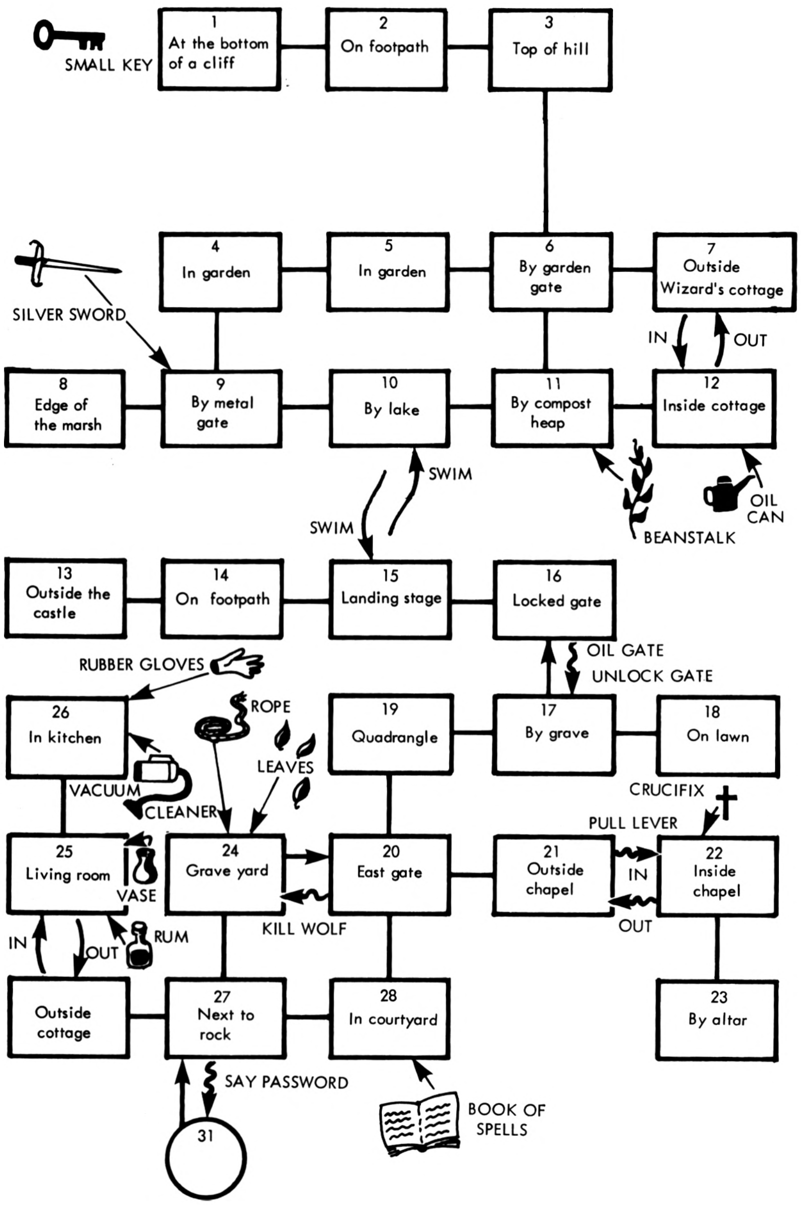
The plot

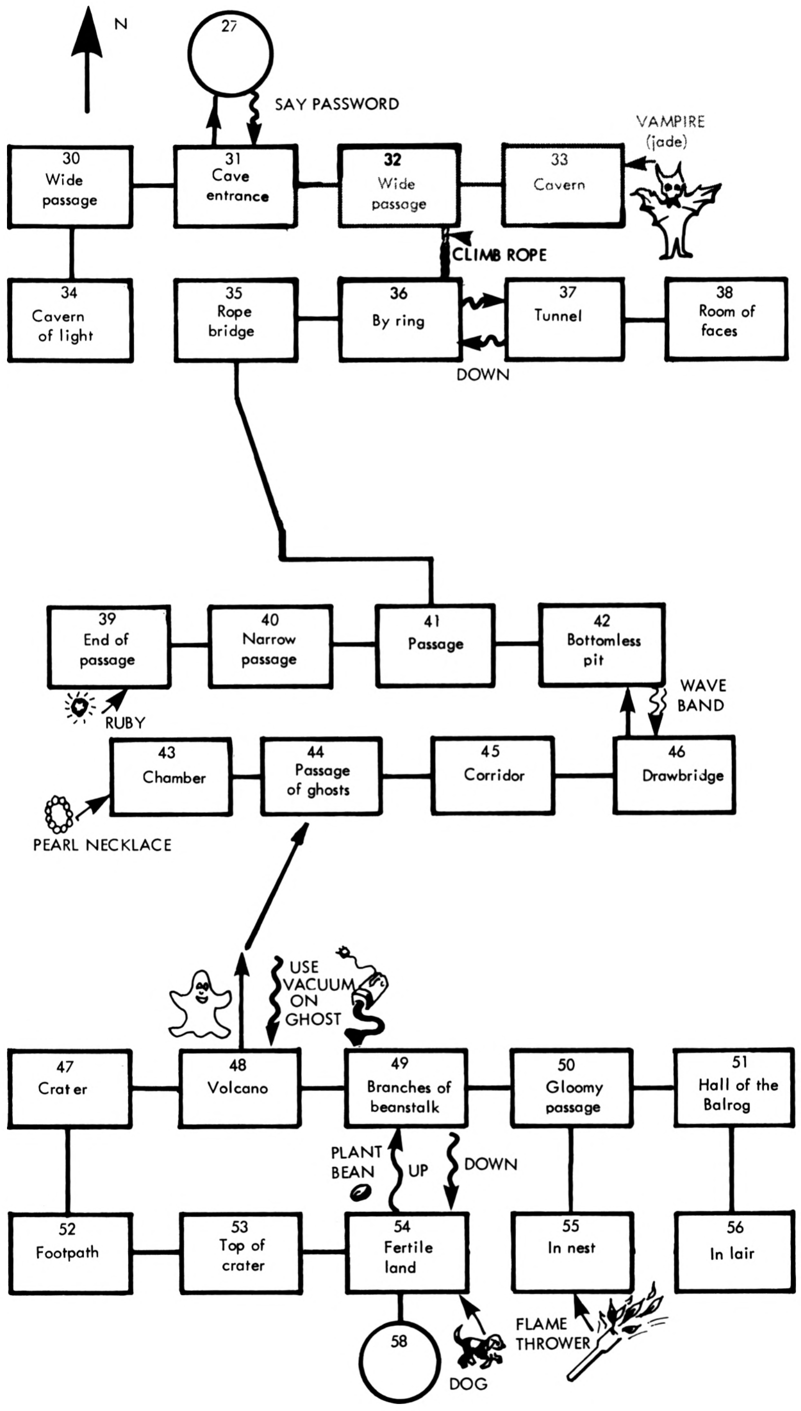
Many years ago, in a land for away, there lived an evil sorcerer who ruled over the whole kingdom. The peasants lived in fear of this cruel and heartless being, who would send his servants late at night

```
What should I do now ?
I am :-
outside a small cottage. A sign on the
door reads 'Wizard out at the moment.
Please leave treasures inside !'.

I can go :-
West,In

What should I do now ?  go in

I am :-
inside the Wizard's cottage. A small
fire burns in the grate.

I can go :-
Out

Things I can see :-
a can of oil

What should I do now ?  get oil

I am :-
inside the Wizard's cottage. A small
fire burns in the grate.

I can go :-
Out

What should I do now ?  inventory
I am carrying :-
a can of oil

I am :-
inside the Wizard's cottage. A small
fire burns in the grate.

I can go :-
Out
```

Fig. 1.4 Sample run of game program.

to take their valued possessions and hide them in the castle high above the village. Only this morning, you received a note from the poor old Wizard asking for your help in recovering the treasures. Will you help him? Can you find the ten items of treasure stolen by the sorcerer and now guarded over by evil creatures and return them to the Wizard's cottage?




N
27
SAY PASSWORD
VAMPIRE (jade)
30 Wide passage
31 Cave entrance
32 Wide passage
33 Cavern
CLIMB ROPE
34 Cavern of light
35 Rope bridge
36 By ring
37 Tunnel
38 Room of faces
DOWN
39 End of passage
40 Narrow passage
41 Passage
42 Bottomless pit
WAVE BAND
RUBY
43 Chamber
44 Passage of ghosts
45 Corridor
46 Drawbridge
PEARL NECKLACE
USE VACUUM ON GHOST
47 Crater
48 Volcano
49 Branches of beanstalk
50 Gloomy passage
51 Hall of the Balrog
PLANT BEAN
UP
DOWN
52 Footpath
53 Top of crater
54 Fertile land
55 In nest
56 In lair
58
DOG
FLAME THROWER

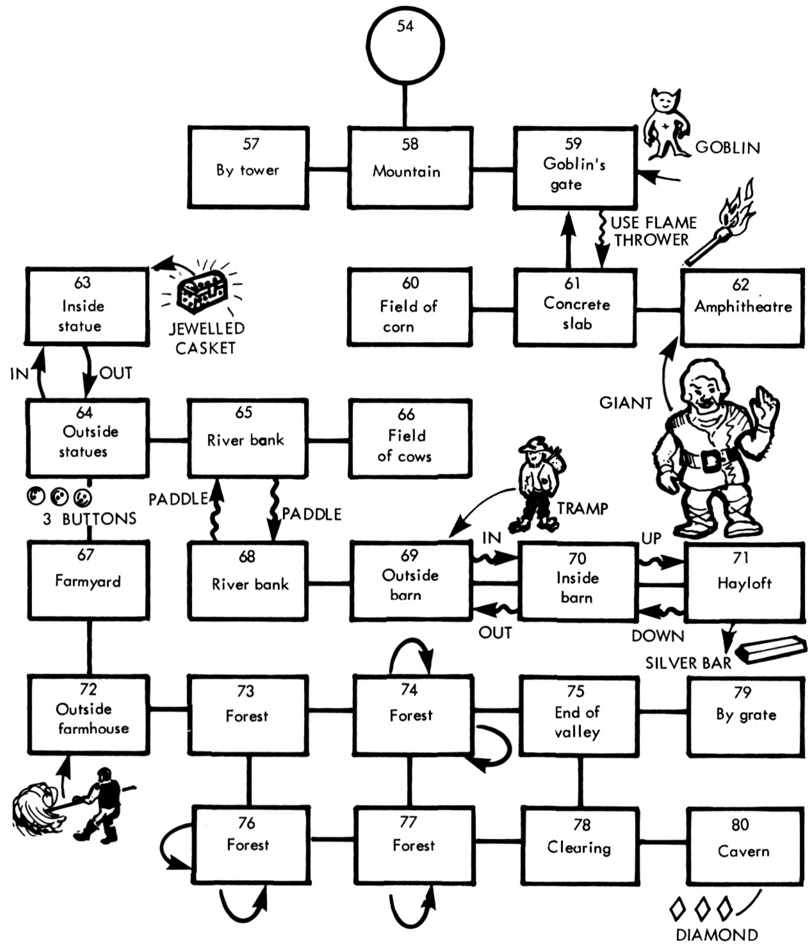


Fig. 1.5 Map for *The Wizard's Quest.*

I have used the same technique to draw the map for this game, but with 80 locations, it was necessary to split it into three sections (Fig.1.5.). I have tried to ensure that there is only one way across from one page of the map to the next, so as to avoid undue complexity. There are 30 objects in the game, although only ten of these are treasures which give a score when returned to the Wizard's cottage. These are listed in the table below.

Treasure	*Object number*	*Location found in*
A vampire	13	33
A giant slug	15	35

A gold nugget	17	56
A bar of silver	18	71
A diamond	19	80
A jewelled casket	20	63
A giant	21	62
A pearl necklace	28	43
A ruby	29	39
A platinum bar	30	57

A vampire and a giant slug as treasure? Surely not! If you turn to the next chapter to look at the listings for the DATA lines, you will see that these items are included and a quick glance at the scoring routine will again show that they are to be treated as treasure! This illustrates one of the tricks used by adventure writers to make the maximum amount of use of memory.

The vampire

When you first reach location number 33, you will see the vampire. Using the crucifix in this location will, obviously, get rid of him and rather than just emptying the contents of the appropriate array element to make it disappear, I have changed the contents of the same element into an item of treasure ... the jade ring. In a similar way, you can change the contents of an array to make it appear as if there are many more objects than the original 30 which were created in the data lines. Obviously, you have to be careful when using this technique that you don't allow the player to carry a vampire around with him! This can be prevented by setting the value of a variable and checking its value whenever you try to pick the object up. In a very similar way, the slug can be killed by pouring salt all over it and when it disappears, it will leave something behind! See if you can guess how to get rid of the giant!

Changed treasures

Original object	*Location*	*Changes into*	*Method*
A menacing vampire	33	A jade ring	Use crucifix
A giant slug	35	A silk purse	Use salt
A giant	62	An emerald	Using the sling

In addition to changing vampires and the two other dangerous creatures into items of treasure, I have also included a few other objects which change their nature during play. The table below lists all the objects found in the games, together with any changes which may occur to them. In future I shall always refer to an object by its number. Hence object number 4 is the vacuum cleaner and object 17 the gold nugget.

Objects within the game

Number	*Original object*	*Location*	*Does it change?*	*What to?*
1	Small beanstalk	11	Yes	Giant beanstalk
2	A can of oil	12	No	
3	A small key	1	No	
4	Vacuum cleaner	26	No	
5	A glass vase	25	No	
6	Rubber gloves	26	No	
7	A magic wand	23	No	
8	A bottle of rum	25	No	
9	A book of spells	28	No	
10	A gleaming sword	9	No	
11	....	24	Yes	A rope and hook
12	A pile of leaves	24	No	
13	An evil vampire	33	Yes	A jade ring
14	A wooden crucifix	22	No	
15	A giant slug	35	Yes	A silk purse
16	A jar of salt	38	No	
17	A gold nugget	56	No	
18	A silver bar	71	No	
19	A diamond	80	No	
20	A jewelled casket	63	No	
21	A giant	62	Yes	A large emerald
22	A flame thrower	55	No	
23	A crowbar	63	No	
24	A row of buttons	64	No	
25	A little dog	54	Yes	Disappears
26	An angry farmer	72	Yes	Disappers
27	....	72	No	A sling
28	A pearl necklace	43	No	
29	A ruby	39	No	
30	A platinum bar	57	No	

Once you've sorted out what objects and creatures are to appear in your game and where they are to be found, it's back to the map to put the finishing touches to it. Although it may seem to be tedious, drawing out sections of the map again can often be worth your while. There will be occasions where you want to make your map 'one way only', where the player can move from one location to another, but not back again. This can be useful in a game where time plays a part or where some means of transport, other than walking, is used. Imagine, for instance, a game in which you reach the summit of a mountain only to find a large eagle perched in the branches of a tree. If you were to climb onto the eagle's back, it may just fly you to a new location and in this particular case, there would be no way back. Many games do in fact use this sort of technique, as it can make the game more difficult for the player. If they have forgotten to take one of the items they need to solve the next problem, then there will be no way back to find it again! This is one of the occasions where I would draw a wiggly line between the two locations. In this game, however, I have not included any 'one way only' movement.

The maze

Another common feature of many adventure games is the maze. There are many ways of drawing the map of a maze when you are designing one to make life more difficult for the player. I have never been fond of mazes in adventure games, finding them dull and boring. This is probably because I lack the patience to solve them. Nevertheless, a book about adventure programming without a maze drawing would be incomplete and therefore I have included a fairly simple one in locations 73, 74, 76 and 77 to illustrate how they can be created. These four locations are all within the dark and gloomy forest, where movement does not obey the normal rules of logic. Movement north from location 76, for example, takes you back through the trees to location 76 again. There's no reason at all why you shouldn't include many more locations within the maze and twist the arrows all over the place to confuse the player! It is, of course, important to make sure that the decriptions of the locations within the maze are all exactly the same, otherwise the player will be able to sort out where they are too easily.

Yet another common feature of an adventure game is being unable to move around freely until a problem or puzzle is solved. In this particular game, for example, there is a ghost in location 44 who will not let you progress further into the game. Once you have sucked the ghost up into the vacuum cleaner, the path is then cleared so that you are free to move south. The trick in programming these sorts of puzzles lies in setting the value of a variable and testing its value whenever the player attempts to move from that location. There are 13 locations in this game where puzzles must be solved before being able to progress further and these are summarised in the table below.

Puzzles to be solved

Location	*Puzzle*	*Solution*
16	The gate is locked	1 Oil the padlock 2 Unlock it with the key
20	The wolf blocks your way	Kill it with the sword
27	No way into the caves	1 Read the password 2 Say the password
36	The cave overhead is too high to reach	1 Throw the rope which will catch on the ring 2 Climb the rope
42	You are at the side of the bottomless pit	You must wave the magic wand
44	The ghost blocks your way south	Use the vacuum cleaner to suck it up
54	There is no way up to the caves above	1 Plant the beanstalk 2 Fill the vase with water 3 Pour the water onto the beanstalk to make it grow 4 Climb the beanstalk
59	The goblins block your way	Use the flame thrower on them
64	The door is closed	Press the correct button
69	The tramp won't let you into the barn	Give him a bottle of spirits
79	The grate is set into the ground	Use the crowbar to prise it open
22	You are locked in the small chapel	Pray for help
21	The chapel door is closed and you can't get in	1 Wear the rubber gloves to protect you from electric shocks 2 Pull the lever and go in

Some of the problems in the game require two or more puzzles to be solved. The way in to the small chapel, for instance, involves pulling the lever. Unfortunately, however, the lever is connected to a high voltage and unless the player is wearing rubber gloves, he will end up dead!

In a few locations the player must adopt a different approach to moving in the normal directions and these are listed below.

Movement

Location	*Method of movement*	*Location reached*
7	Go in	12
12	Go out	7
10	Swim across	15
15	Swim across	10
36	Climb the rope (after it's been thrown)	37
37	Go down the rope	36
29	Go in	25
25	Go out	29
54	Go up the beanstalk (if it has grown)	49
49	Go down the beanstalk	54
64	Go in (after pressing button)	63
63	Go out	64
65	Paddle across	68
68	Paddle across	65
69	Go in	70
70	Go out	69
70	Go up the ladder	71
71	Go down the ladder	70
79	Go down (after using the crowbar to open the grate)	80
80	Go up	79

All the above information has been shown on the map and you should be almost ready to move over to the keyboard to start the coding.

The only other decision which needs to be made is whether to include sound or graphics within the game. Even the simplest of sound effects can transform a game beyond all recognition if the sounds are relevant to the game, but all too often sound is tacked on as an afterthought and does nothing to improve the game. The sound of a person knocking on a door, a radio playing in the corner of the room or even a ghostly scream can add the finishing touches to a game, although the programming of such effects will probably take you a long time.

Good graphics can perform even greater miracles, but a graphics adventure needs to be planned as such right from the start; inserting pictures as an after thought is unlikely to be very successful. In *The*

Wizard's Quest, I have included neither graphics nor more than a few simple sounds and have concentrated on the basic essentials, although in the subsequent two games, graphics and sound play an integral part of the program. Many of the sections of coding of a pure text adventure can be used to equal effect in a graphics game and if you compare the sections of each game, you will find many similarities.

2 Writing the data

Your first task when converting your map into the data for your game is to choose the names for the variables you will use. At this point, you will need only to choose the names of the variables associated with the arrays. The others can be selected as the game is developed. There are a number of considerations which need to be taken into account when choosing the names of your variables. Many programmers argue that long variable names help you to remember their purpose and there is little doubt that they do help to make a program easier to follow. Thus using map%(2,3) and locations$(7) will immediately remind you of their purpose, but using such long variable names does use rather more memory than is absolutely necessary. For this reason, I have used single letter variable names for all variables in the games in this book. If you do decide to use long variable names, do remember that they must *not* contain keywords. Thus place$ would be valid, whereas locate$ would not!

One point worth noting about Locomotive BASIC is that the variables can be in lower or upper case, but will be treated as being *exactly* the same. Unlike many home computers, the Amstrad does not convert variables typed in lower case into upper case, although it does treat them as being the same. Thus it doesn't matter whether you type in A$(23) or a$(23).

The second, and probably the most important, consideration when choosing variables is to use integer variables wherever possible. They use only a fraction of the RAM needed to store real variables. In adventure games, we will normally be dealing with whole numbers and therefore can make widespread use of integer variables. Defining variables as integers can be done at the start of the program using the DEFINT command, but in this book, I've used the % sign at the end of the variable name in preference. The choice is up to you!

The final consideration when choosing names, is purely one of convenience. If you intend to write just one adventure game, it really doesn't matter what name you give to your variables, but if you decide that you are going to write several games, then it makes sense to stick to the same names in each of your games. In a sense, therefore, the names of variables can be seen as a trademark of the programmer and if you look through any computer magazine, you can often recognise the author of an adventure listing by the names of the variables used in the game.

The following list contains the names of the major variables used in all three games.

Variable name	*Purpose*
P%	Holds the player's current position
S%(X,Y)	Holds the map of the game
Q$(X)	Holds the descriptions of the objects found in the game
B%(X)	Holds the location where the object is to be found
N$(X)	Holds the word by which the computer recognises the object
N%(X)	Holds the pointer to which object has been mentioned by the player
V$(X)	Holds the descriptions of the objects being carried by the player (the inventory)
A(X)	Holds the flag to test if carrying a particular object

The most useful mode for adventure games on an Amstrad computer is the 40 column mode (MODE 1). The text in an 80 column mode is not very easy to read on the colour monitor, and in the 20 column mode, you can't fit many words on the screen. In this game, I have selected the standard default colours of MODE 1, red, blue, yellow and cyan. Rather than typing in the INK settings for each of these colours, I have used a call to achieve this (CALL &BCO2).

```
10 REM ** The Wizard's Quest **
20 REM ** an adventure for the Amstrad CPC464 **
30 REM ** Steve Lucas   1985 **
40 MODE 1
50 PEN 1:LOCATE 10,2:PRINT"The Wizard's Quest"
60 LOCATE 4,10:PEN 1:PRINT"An adventure game by S.
W. Lucas"
70 REM ** common messages **
80 Y$="O.K."
90 CALL &BC02
```

```
100 DIM Q$(80),S%(80,4),G$(30),B%(30),N$(30),N%(30
),V$(4),A(30)
110 REM ** READ the DATA for the locations **
120 FOR X=1 TO 80: READ Q$(x)
130 FOR Y= 1 TO 4: READ S%(X,Y)
140 NEXT Y,X
150 DATA standing in a small gully at the bottom o
f a sheer cliff face.,0,0,2,0
160 DATA on a narrow footpath between two steep  m
ountains.,0,0,3,1
170 DATA at top of a small wooded hill. A narrow f
ootpath leads west into the mountains.,0,6,0,2
180 DATA on a dirt track which winds its way     t
hrough a well tended garden.,0,9,5,0
190 DATA walking through a garden which is full  o
f beautiful flowers.,0,0,6,4
200 DATA by a garden gate. The path to the north l
eads out of the garden into open       countryside
.,3,11,7,5
210 DATA outside a small cottage. A sign on the  d
oor reads 'Wizard out at the moment.   Please leav
e treasures inside !'.,0,0,0,6
220 DATA on the edge of a marsh. An old sign herer
eads 'Danger...do not proceed west!'.,0,0,9,0
230 DATA by a large wooden gate. Strange runes   a
re inscribed on it.,4,0,10,8
240 DATA on the shores of a small lake. A small  i
sland lies in the middle.,0,0,0,9
250 DATA by the compost heap. A few small        b
eanstalks  are growing out of the top  of the heap
.,6,0,0,0
260 DATA inside the Wizard's cottage. A small    f
ire burns in the grate.,0,0,0,0
270 DATA outside a gloomy castle. There appears  t
o be no way in.,0,0,14,0
280 DATA on a footpath lined with dense shrubs.,0,
0,15,13
290 DATA on a small landing stage. A few boats   a
re moored here.,0,0,16,14
300 DATA at the entrance to a disused graveyard. A
 rusty chain is padlocked around the   two metal g
ates.,0,0,0,15
310 DATA standing next to an old gravestone. It'se
ngraved with the message 'To Martha....Please Help
 Me !!',16,0,18,19
320 DATA standing on a small lawn with a tall    h
edge on three sides.,0,0,0,17
330 DATA in a small quadrangle full of ancient   t
ombstones.,0,20,17,0
340 DATA by the East gate. A howling wolf guards t
he way west.,19,28,21,0
350 DATA outside a small chapel. The door is     c
losed at the moment. A large lever     protrudes f
rom the wall.,0,0,0,20
```

```
360 DATA inside an ornate chapel. The door has   c
losed behind me.,0,23,0,0
370 DATA next to the altar. There is nobody here.,
22,0,0,0
380 DATA outside the graveyard. A path leads     s
outh and down from here.,0,27,20,0
390 DATA in the living room. The old woodcutter  i
s asleep in a chair.,26,0,0,0
400 DATA in a small kitchen. The sink is full of d
irty pots.,0,25,0,0
410 DATA next to a large rock which blocks the   e
ntrance to a cavern. Strange runes are engraved on
 it.,24,0,0,29
420 DATA in a small courtyard full of old bones. T
he only way out is north.,20,0,0,0
430 DATA outside a wooden cottage. A sign on the d
oor reads 'Woodcutter for Hire'.,0,0,27,0
440 DATA in a wide passage lit by a strange greeng
low coming from the south.,0,34,31,0
450 DATA standing in the entrance to the Caverns o
f the Xarda.,27,0,32,30
460 DATA in a wide east-west passage. A smaller  p
assage leads south and down from here.,0,36,33,31
470 DATA in an enormous cavern which is lined    w
ith grotesque faces.,0,0,0,32
480 DATA in the 'Cavern of Light'. A large       c
rystal hangs in the centre and sends   rays of gre
en light dancing along the   walls.,30,0,0,0
490 DATA on a narrow rope bridge which spans a   d
eep underground gully.,0,41,36,0
500 DATA in a small cavern. There is a tunnel    h
igh above me leading east. A large     metal ring
hangs from the ceiling next  to the tunnel.,32,0,0
,35
510 DATA in a gloomy tunnel which looks down intoa
 small cavern. There is a rope hanging down from h
ere.,0,0,38,0
520 DATA in the 'Room of Many Faces'. The walls  a
re lined with mirrors which reflect    thousands o
f images of my face.,0,0,0,37
530 DATA at the end of a passage leading into them
ountain. The view over the valley is   magnificent
.,0,0,40,0
540 DATA in a narrow east-west passage lit by a  b
eam of daylight.,0,0,41,39
550 DATA in a narrow east-west passage. To the   n
orth lies a rope bridge which stretchesacross a de
ep ravine.,35,0,42,40
560 DATA at the edge of the bottomless pit. A    d
rawbridge can be seen on the southern  side.,0,0,0
,41
570 DATA in a chamber full of furniture built fors
omeone who must be extremely small.,0,0,44,0
580 DATA at the end of a wide passage.An evil    g
```

```
host stands guard and prevents me     passing sou
th.,0,0,45,43
590 DATA in a wide east-west passage lit by      t
orches high above my head.,0,0,46,44
600 DATA on a wooden drawbridge.,42,0,0,45
610 DATA in the crater of an extinct volcano.,0,52
,48,0
620 DATA in a small passage. Daylight pours into t
he passage from an opening to the west.,44,0,0,47
630 DATA in the branches of a giant beanstalk.   T
here is a small cave entrance to the   east.,0,0,5
0,0
640 DATA at the entrance to a gloomy passage.    T
he beanstalk prevents much light       entering.,0
,55,51,49
650 DATA in the 'Hall of the Evil Balrog'. The   w
alls are all scorched.,0,56,0,50
660 DATA on a footpath leading between the centrea
nd the top of the crater.,47,0,53,0
670 DATA at the top of the crater. A path leads  e
ast and down the mountainside.,0,0,54,52
680 DATA on a path leading down the mountainside.I
 can see a cave entrance in the cliff  high above
my head. The soil here is    very fertile!,0,58,0,
53
690 DATA in the nest of the Balrog. Three        e
normous eggs lie at the centre.,50,0,0,0
700 DATA in the Balrog's lair. A tunnel leads    s
outh but the heat and smell coming fromit is too g
reat for me!,51,0,0,0
710 DATA outside the 'Tower of Darkness'. The    e
ntrance is blocked by a pile of rubble.,0,0,58,0
720 DATA on a bracken covered hillside.,54,0,59,57
730 DATA by the 'West Gate of Jadir'. Two vicioush
obgoblins stand guard.,0,0,0,58
740 DATA in a field of golden corn.,0,66,61,0
750 DATA on a large strip of concrete. To the    n
orth lies the West Gate.,59,0,62,60
760 DATA in an amphitheatre. A giant flexes his  m
uscles at the far side.,0,0,0,61
770 DATA inside the bronze statue. A two headed  l
izard with halitosis peers down at me  from above.
,0,0,0,0
780 DATA on the banks of a river.It is too deep  h
ere to cross. A seventy foot bronze    statue of t
he god Joliar stand here.,0,67,65,0
790 DATA on the banks of a shallow river. It     l
ooks safe to cross here.,0,0,66,64
800 DATA in a field full of cows grazing.,60,0,0,6
5
810 DATA in a farmyard. An old dog sleeps in the s
hade of the haystack.,64,72,0,0
820 DATA on the banks of a shallow river. A sign h
ere reads 'Danger Quicksand...don't go west!',0,0,
69,0
```

```
830 DATA outside an old barn. A path leads west. A
n old tramp blocks my way in.,0,0,0,68
840 DATA inside the old barn. A rickety ladder   l
eads up into the hayloft.,0,0,0,0
850 DATA in the hayloft. A cat lies in the hay   s
leeping.,0,0,0,0
860 DATA outside an old farmhouse. The door is   l
ocked and there is no way in. A        footpath le
ads west into the forest.,67,0,73,0
870 DATA in a dark and gloomy forest.,73,76,74,72
880 DATA in a dark and gloomy forest.,74,77,74,73
890 DATA at the west end of a small valley.,0,78,7
9,0
900 DATA in a dark and gloomy forest.,73,76,77,76
910 DATA in a dark and gloomy forest.,74,77,78,76
920 DATA in a small clearing. The way north leadsi
nto open countryside.,75,0,0,77
930 DATA at the far end of the valley. A metal   g
rate is set into a concrete slab in theground here
.,0,0,0,75
940 DATA in a small hole under the ground. It is f
ull of soft cushions.,0,0,0,0
```

Line

40	select the 40 column mode
50–60	titles
80	define common message
90	call routine in rom to select the default colours
100	dimension the arrays
110–140	read the data for the locations and the map into the arrays
150–940	data for the locations and the map

Like most versions of BASIC, reading DATA into arrays is very inefficient in the way it uses memory in an Amstrad microcomputer. As you will see later, there are a number of ways of improving this.

In additon to the variables which are common to all adventures, you will probably want to define some variables to contain common messages such as 'O.K.' or 'I can't do that!' This is done in line 80. In this game, I have used just one common message, but you can add extra messages here if you wish!

As there are 80 locations and 30 objects in this game, line 100 is used to dimension the arrays large enough to hold this information. Thus the array Q$(X) is set to hold the description of 80 locations, whilst S%(X,Y) is set to hold 80 locations and four directions. It is this array which controls the movement of the player from one location to another. The first number in this array refers to the number of the location and the second number refers to the direction. Thus:

S%(27,1)	Refers to the location reached by going north from location 27
S%(27,2)	Refers to the location reached by going south from location 27
S%(27,3)	Refers to the location reached by going east from location 27
S%(27,4)	Refers to the location reached by going west from location 27

The array V$(X) is used to hold the descriptions of the objects carried by the player and in this game we are going to limit the number of objects which the player can carry to four. This can be increased or decreased to suit yourself, although you will need to change the number in the inventory, get and drop routines as well.

The best course of action at this stage is to go ahead and type in the program up to line 140. If, after typing in this section, you try to RUN it, the computer will display an error indicating that you are OUT OF DATA. Far from being a nuisance, this error message can be one of the most useful methods of checking out the program as you develop it.

Each of the following 80 lines contains the data for one location. This is in the form of a description followed by four numbers. The first thing you'll notice if you compare the description of location number one in line 150 with the description of the same location on the map is that I've tried to make it far more detailed. One of the major differences between a really good adventure game and a poor one is in the quality of the description. The more detail you can include within the description, the more vivid the picture built up in the mind of the player. If you compare the two examples of screen displays from text adventures, you'll see what I mean.

Example 1

```
I am in a gully
I see a key
What shall I do now?
```

Example 2

```
I am standing in a small gully at the bottom of a sheer rock face. The
leaves from a tree keep falling on my head.
I can go east
Things I can see:-
a small rusty key
What shall I do now?
```

Example 2 provides the player with more information about his location, and this in turn leads to a far greater sense of involvement for the player. With 43K of RAM available for use in BASIC, there's no excuse for descriptions of locations and objects which are so brief that they fail to give the player enough information. On the other hand, don't be tempted to go overboard with the text, or you'll soon run out of memory. There are a number of ways of compressing extra detail into the data, but these techniques are beyond the scope of this book.

To convert the map for your game into the data lines for the program you must examine each location in turn. The four numbers in each data line correspond with the number of the location reached by travelling north, south, east and west from the location in question. As an example of this, consider location 4 in *The Wizard's Quest.* To make life easier, I have kept the data for each location on a separate data line. Thus the data for location 4 is found in line 180, the fourth data line. In this location, the player cannot travel north or west, and hence the first and fourth numbers after the description must be zero, to indicate that movement north and west is not possible. Movement south from this location takes you to location 9, whilst movement to the east takes you to location 5. The second and third numbers in the data line would, therefore, be 9 and 5 respectively. When these four numbers are read into the array S%, which holds the map, the contents of the fourth element of S% would be as follows:

S%(4,1) = 0 Movement north from location number 4 is not possible
S%(4,2) = 9 Movement south from location number 4 takes you to location 9
S%(4,3) = 5 Movement east from location number 4 takes you to location 5
S%(4,4) = 0 Movement west from location number 5 is not possible

In this game there are a few locations where the player can go up or down in addition to movement in the normal compass directions. As already indicated, this movement can be dealt with by storing the number of the location reached by going up or down in the data lines for reading into the array S%. I have not done this because there are too few locations where the player can go up or down to justify the extra memory space used by increasing the dimension of the array. However, you may like to experiment with this and the example below should indicate how to set about it. If you would like to try your hand at a three dimensional game, there is more information on the subject in Chapter 16.

Supposing movement up from location 1 took you to location 7 and movement down from location 1 to location 19, then the first data line would need to be changed to:

150 standing in a small gully at the bottom of a sheer cliff face.,0,0,2,0,7,19

You would probably need to change the description of the location so that going up or down sounds more feasible.

```
150 standing in a small gully at the bottom of a sheer cliff face. A narrow path leads up the mountainside and a small grate leads down into a dark tunnel.,0,0,2,0,7,19
```

Adding the extra numbers to each data line without changing the routine to READ them into the array would result in disaster. You would need to change line 130 to:-

```
130 FOR Y=1 TO 6:READ S%(X,Y)
```

and also have to add a number of extra lines later in the game.

Testing
As soon as you have entered all the data lines for the locations in the game (lines 150 – 940), you will need to check whether they have been typed in correctly or not. Even the slightest typing error at this stage can cause problems, especially to a beginner. Over ninety per cent of errors occurring at this stage will be due to commas in the wrong place, and the worst thing about a mistake in the placement of commas is that the computer will, more often than not, tell you that an error has occurred in a place other than where the real mistake lies! The easiest method of checking that you have made no mistakes may seem to involve sitting down and checking the listing on the screen of your computer with that printed on paper. In practice, however, you are more likely to miss the mistakes, especially when you are getting towards the end and are tired and frustrated. The best approach is to try to RUN the program. If all is well, and the computer has found no errors in the order of the data, the titles will be printed on the screen and after a few seconds the message 'Ready' will appear on the screen. A message to this effect does not mean that there are no mistakes in your program, but merely that the computer has been unable to find any. A computer is unable to check whether you have made a spelling mistake in the description of a location or whether you have inserted the number of the correct location when going north from location 17. It can, of course, tell you if the data is presented in the wrong order or if a comma is missing.

If the computer does return with a SYNTAX ERROR or an OUT OF DATA error, the computer has found an error in the data lines, even if it tells you that the error occurs in line 120 or 130. Tracking down the source of the mistake requires a little thought on your part. The easiest way is to try to find out the number of the location where the computer thinks that the error occurs by typing PRINT X and pressing ‹ENTER›. The number printed on the screen will probably correspond with the number of the location where the error has occurred. If you want to make sure of this, you should ask the computer to print the description of the previous location.

For example, supposing the computer prints the value of X = 19, then the last correct location would be number 18, so typing PRINT Q$(17) should print the correct description of location 17. If the description of location 17 is correct, try typing PRINT Q$(18) to see if the description of location 18 has been read correctly. By a process of careful elimination, you should be able to track down the position in the data lines where the actual error has occurred. Once the mistake has been rectified, you should try running the program again until, eventually, the program will run without an error being discovered by the computer.

One point that's worth looking out for at this stage is that you have not included commas in the descriptions of any of the locations in the DATA lines. The listings in this book do not have quotation marks around these descriptions because, in most cases, they are unnecessary. If, however, you want to include a comma in the description of any location, then you *must* enclose that description in quotation marks, or the computer will interpret the comma as being the start of a new item of data which will result in a syntax error at some other point in the program. For example,

```
250 DATA by a compost heap. A few small, green beanstalks have
grown out of the top of the heap.,6,0,0,0
```

would produce an error and should be written as:

```
250 DATA "by a compost heap. A few small, green beanstalks are
growing out of the heap.",6,0,0,0
```

Note, however, that if you do want to include a description of a location which does contain a comma within it and this description is saved as a data file in the routine to save a game, then it may cause confusion when the tape is read in again. You would, therefore, be well advised to make sure that your descriptions don't contain commas.

Even if the computer doesn't find an error for you, you could still have made a simple spelling mistake. It really does pay you to double check all the details at this stage, rather than waiting until later, when it will be much harder to find errors. The easiest method of checking the data is to RUN the program and, when it prints the message 'Ready', type in the short line below and press ‹ENTER›.

```
FOR X=1 TO 80: PRINT X:PRINT Q$(X): NEXT X        ‹ENTER›
```

This will print out the numbers from 1 to 80 and, alongside each, will print the description of the appropriate location. Pressing the escape key will make the computer pause to give you time to read the screen and check the accuracy of the descriptions. Spelling mistakes in adventure games can spoil an otherwise excellent game, and you should check and double check each description. Don't forget to use a dictionary if you, like me, tend to be poor at spelling! Equally irritating to an adventure player is a description where words are split over two lines. As we are using a 40 column screen width, the descriptions have been adjusted, by inserting extra blank spaces, so that no word is started after column 37 and that there are no split words. Even professional games sometimes contain errors of this type and they can be very difficult to track down if you leave the task until later.

The final check to be make at this stage is that the DATA entered for the map contains no errors. There are two ways of checking this information, either by checking the listing very carefully, line by line or by typing PRINT S%(1,1) etc. and checking that the value returned agrees with your map. Whichever method you choose, it will take time, but you won't regret it later! It is all too easy to miss a simple mistake in one data line and just one number which is incorrect can make the whole map of the game appear nonsense!

After keying in so much of the program into your computer, you will probably be feeling tired , and if you continue entering you will be much more likely to make errors than when you first started. The best course of action at this stage is to take a break, but don't forget to save a copy of the game onto tape or disc before leaving the keyboard. There's nothing worse than leaving the keyboard for five minutes and returning to find that the kids have loaded the latest arcade game or that the cat has knocked the computer off the desk and the plug has come out. In fact I'd strongly recommend that you make a habit of saving your program every half hour or so. You never can tell when disaster is likely to strike and if you adopt this course of action, you'll never lose more than half an hour's work, even if the worst does happen! If you do have a disc drive, then I would suggest that you save a copy of your new version using a different file name, so that you will then have two, or more, copies to fall back on.

Reading the data for the objects

```
950 REM ** READ the DATA for the OBJECTS **
960 FOR X=1 TO 30:READ G$(X),B%(X),N$(X):N%(X)=X:
NEXT
970 DATA a small beanstalk,11,beanstalk,a can of o
il,12,oil,a small key,1,key
980 DATA a vacuum cleaner,26,vacuum,a glass vase,2
5,vase,a pair of rubber gloves,26,gloves
990 DATA a magic wand,23,wand,a bottle of rum,25,r
um,a book of spells,28,book,a gleaming sword,9,swo
rd
1000 DATA "",24,"",a pile of leaves,24,leaves,a me
nacing vampire,33,vampire
1010 DATA a wooden crucifix,22,crucifix,a giant sl
ug,35,slug,a jar of salt,38,salt
```

```
1020 DATA a ** GOLD NUGGET **,56,gold,a ** BAR OF
SILVER **,71,silver,a ** DIAMOND **,80,diamond
1030 DATA a ** JEWELLED CASKET **,63,casket,a gian
t,62,giant,a flame thrower,55,flame
1040 DATA a crowbar,63,crowbar,a row of three butt
ons,64,buttons,a friendly dog,54,dog
1050 DATA an angry farmer,72,farmer,"",72,"",a **
PEARL NECKLACE **,43,pearl
1060 DATA a ** RUBY **,39,ruby,a ** PLATINUM BAR *
*,57,platinum
```

Line

960	read the description of the object, the location where the object is found and the word it is recognised by for each of the 30 objects. Also set the pointer N%(X) to equal the number of the object.
970–1060	data for the 30 objects.

The section of code between lines 950 and 1060 is used to READ the DATA for the 30 objects found in the game. If your game contains more than 30 items, then you will need to increase the size of the arrays in the DIM statements at the start of the program and also change the size of the loop in line 960. Each line of DATA contains the information for several objects, so as to pack as much information into the game as possible. This data is in three parts : the description of the object, its location and the word which the computer will recognise it by. The final array (N%(X)) is set to act as a pointer to the number of the object. Although this section of the program is much shorter than the previous one and won't take you as long to enter into your computer, it is just as important to check that the computer is READing the DATA for the objects into the array correctly. This can be done by trying, once again, to run the program and checking that the computer doesn't find any errors. Should an error occur at this stage, you should type PRINT X and press ‹ENTER›. The value of the variable X will probably indicate the number of the object where the error has occurred. By a process of careful elimination, you should be able to track down the exact source of the mistake and correct it. If all is well, do check through all the variables to make sure that no spelling mistakes have crept in. The following line should help you to do this:

```
FOR X=1 TO 30: PRINT X: PRINT G$(X): PRINT N$(X): NEXT X
```

The computer will print the description of all of the objects, with the exception of the few which are initially undefined (as explained in Chapter 1), together with the words which the player will have to type in. One very common mistake, which can happen if you try to be too quick when typing, is that you get the line number wrong.

Imagine that you are in a hurry and type line 1020 as line 102, or even worse as line 120. In the first case, the data for the game will be in the wrong order and will appear to spoil the section you had previously checked. In the second case, you would actually have typed an incorrect line to replace the original line 120. It really does pay you to check each line before actually entering it into the computer's memory. When you are sure that all is well, you should save a copy of the new version of your game onto tape or disc before switching off.

The main control section 3

In any adventure, the most important section of all is the main control sequence. In principle, this is a fairly straightforward piece of programming, but unless it is carefully planned it can become much more complex than it really needs to be. The simpler the structure of the control section, the easier it is to detect any errors, and this in turn helps to keep program development times to the minimum. Many different approaches may be adopted when writing this section, although once you've found a method that suits you you will probably wish to stick to it in future games. The fun and enjoyment of writing adventure games comes from setting devious problems for the player to puzzle over rather than from spending many hours developing routine sections of code. There is nothing guaranteed to dampen the creative spirit more than spending many hours debugging routines which are more complex than they really need to be, especially when you are eager to put your ideas for puzzles into practice.

We have already seen how the control sequence fits into the framework of the game and we now need to sort out its internal structure. The best way of doing this is to draw yet another flowchart (Fig 3.1).

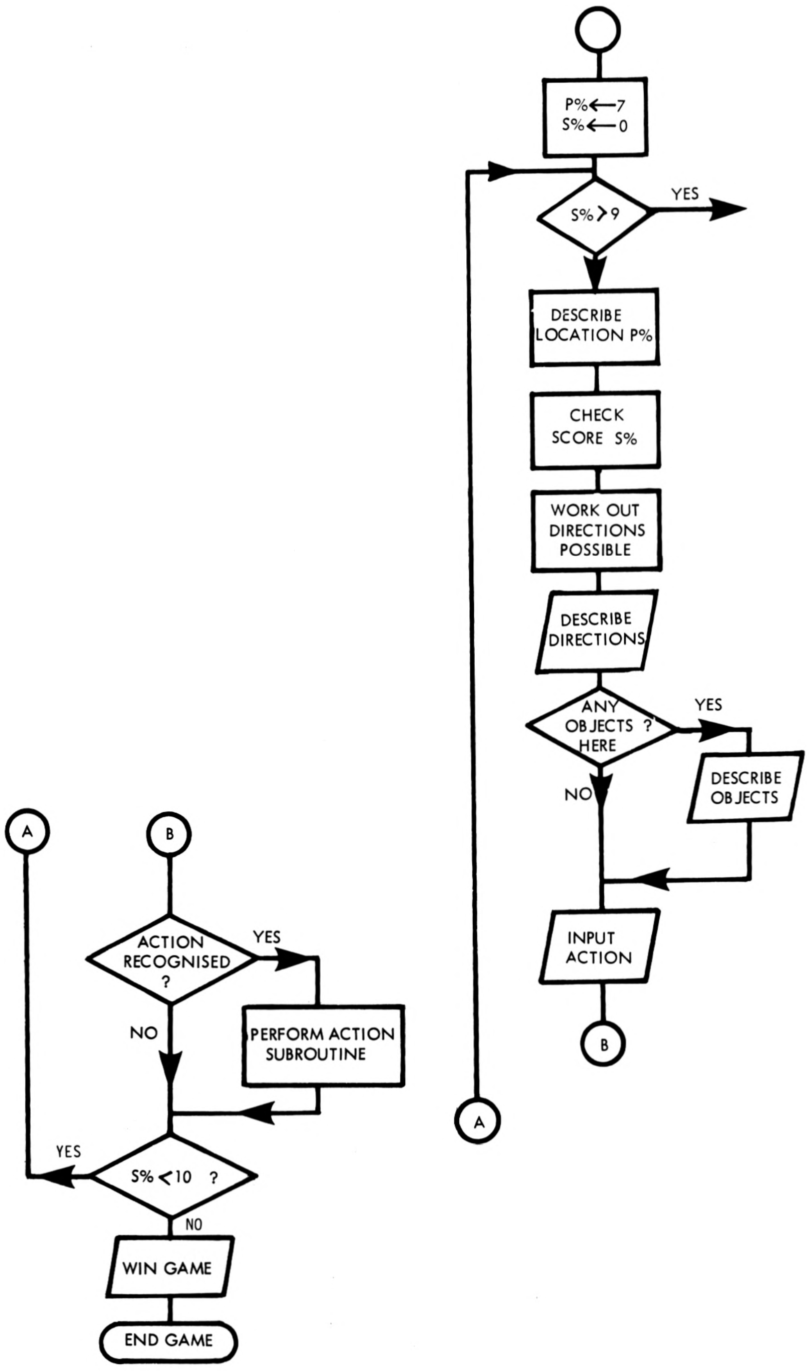


Fig. 3.1 Flowchart for control sequence.

The listing below, from *The Wizard's Quest,* shows how I have converted the flowchart into a working routine, capable of controlling the game.

```
1070 REM ** set starting position and score **
1080 P%=7:S%=0:CLS
1090 REM ** main control loop **
1100 WHILE S%<10
1110 PRINT:PEN 1:PRINT"I am :- ":PEN 2:PRINT Q$(P%
)
1120 REM ** check score **
1130 GOSUB 2000
1140 REM ** describe directions **
1150 PEN 1:A$="":IF S%(P%,1)>0 THEN A$="North"
1160 IF S%(P%,2)>0 AND LEN(A$)>0 THEN A$=A$+",Sout
h" ELSE IF S%(P%,2)>0 THEN A$="South"
1170 IF S%(P%,3)>0 AND LEN(A$)>0 THEN A$=A$+",East
" ELSE IF S%(P%,3)>0 THEN A$="East"
1180 IF S%(P%,4)>0 AND LEN(A$)>0 THEN A$=A$+",West
" ELSE IF S%(P%,4)>0 THEN A$="West"
1190 IF (P%=69 AND SH=1) OR P%=7 OR P%=21 OR P%=29
 OR P%=64 THEN A$=A$+",In"
1200 IF P%=12 OR P%=63 THEN A$="Out" ELSE IF P%=22
 OR P%=25 THEN A$=A$+",Out"
1210 IF P%=70 THEN A$="Up,Out" ELSE IF P%=80 THEN
A$="Up"
1220 IF P%=54 AND SL=1 THEN A$=A$+",Up"
1230 IF P%=37 OR P%=49 OR P%=79 THEN A$=A$+",Down"
 ELSE IF P%=71 THEN A$="Down"
1240 IF A$="" THEN A$="nowhere obvious!"
1250 PEN 1:PRINT:PRINT"I can go :-":PEN 2:PRINT A$
1260 PEN 1:PRINT:
1270 REM ** describe objects **
1280 E=0:FOR T=1 TO 30
1290 P=0:IF B%(T)=P% THEN P=1
1300 IF P=1 THEN 1320
1310 NEXT T:GOTO 1340
1320 IF E=0 THEN PRINT"Things I can see :-":PEN 2
1330 PRINT G$(T):E=1:GOTO 1310
1340 PRINT:PEN 1:INPUT"What should I do now ";Z$
1350 REM ** analyse input and act on it **
1360 Z$=LOWER$(Z$):B$=LEFT$(Z$,2):C$=LEFT$(Z$,3):D
$=LEFT$(Z$,4)
1370 PRINT CHR$(7):CLS
1380 IF C$="out" OR D$="go o" THEN GOSUB 1800
1390 IF C$="pra" THEN GOSUB 1870
1400 IF C$="in" OR D$="go i" THEN GOSUB 1930
1410 IF (B$="n" OR D$="go n") AND S%(P%,1)>0 THEN
P%=S%(P%,1):PRINT"O.K." ELSE IF (B$="n" OR D$="go
n") THEN PRINT"I can't do that!"
1420 IF (B$="s" OR D$="go s") AND S%(P%,2)>0 THEN
P%=S%(P%,2):PRINT"O.K." ELSE IF (B$="s" OR D$="go
s") THEN PRINT"I can't do that!"
1430 IF (B$="e" OR D$="go e") AND S%(P%,3)>0 THEN
P%=S%(P%,3):PRINT"O.K." ELSE IF (B$="e" OR D$="go
e") THEN PRINT"I can't do that!"
1440 IF (B$="w" OR D$="go w") AND S%(P%,4)>0 THEN
P%=S%(P%,4):PRINT"O.K." ELSE IF (B$="w" OR D$="go
w") THEN PRINT"I can't do that!"
1450 IF C$="sco" THEN PRINT"You have scored ";S%;"
 out of 10."
1460 IF C$="get" OR C$="tak" OR C$="gra" THEN GOSU
B 2130
1470 IF C$="inv" THEN GOSUB 2300
1480 IF C$="dro" OR C$="lea" OR C$="put" THEN GOSU
B 2370
1490 IF C$="wea" THEN GOSUB 2460
1500 IF C$="pul" THEN GOSUB 2520
1510 IF C$="wav" THEN GOSUB 2560
1520 IF C$="pad" THEN GOSUB 4150
1530 IF C$="rea" THEN GOSUB 2670
1540 IF C$="say" OR C$="tal" OR C$="rep" THEN GOSU
B 2710
1550 IF C$="att" OR C$="kil" OR C$="sta" THEN GOSU
B 2780
1560 IF C$="sea" THEN GOSUB 2830
1570 IF C$="thr" THEN GOSUB 2860
1580 IF C$="cli" THEN GOSUB 2930
1590 IF C$="up" OR D$="go u" THEN GOSUB 3010
1600 IF C$="dri" THEN GOSUB 3070
1610 IF C$="giv" THEN GOSUB 3110
1620 IF C$="use" OR C$="pri" THEN GOSUB 3180
1630 IF C$="swi" THEN GOSUB 3420
1640 IF C$="unl" THEN GOSUB 3480
1650 IF C$="oil" THEN GOSUB 3540
1660 IF C$="pla" THEN GOSUB 3580
1670 IF C$="fil" THEN GOSUB 3630
1680 IF C$="pou" THEN GOSUB 3660
1690 IF C$="dow" THEN GOSUB 3720
1700 IF C$="pre" THEN GOSUB 3790
1710 IF C$="hel" THEN PRINT"I'm sorry I don't have
 a clue!"
1720 IF C$="sav" THEN GOSUB 3870
1730 IF C$="loa" THEN GOSUB 4010
1740 WEND
1750 REM ** win the game **
1760 CLS:PEN 1:LOCATE 10,5:PRINT"W e l l   D o n e
!"
1770 PEN 3:LOCATE 3,10:PRINT"You have found and re
covered all the    treasure."
1780 ENV 1,100,2,2:ENT 1,100,-2,2:SOUND 1,284,200,
1,1,1:PEN 2:LOCATE 1,20:PRINT"Goodbye. Thank you f
or playing.":END
```

Line	
1080	set the player's starting postion to location 7 and their score to zero. The screen is then cleared.
1100–1740	this main loop is repeated until the score is equal to ten.
1110	describe the player's current location.

1130	call the subroutine to check the score.
1150–1180	examine the array S%(X,Y) to see if movement north, south, east or west is possible and store this information in the variable A$.
1190–1230	check the number of the location to see if movement up, down, in or out is possible and add this information to A$.
1250	describe the direction in which the player can travel.
1260	print a blank line.
1280–1310	check all of the thirty objects to see if they are in the current location.
1320	if this is the first object in that location, print the message 'Things I can see :-'.
1330	describe the objects found in the current location.
1340	input the player's instructions.
1360	change the sentence into lower case letters and examine the first letters of it.
1370	clear the screen and make short note.
1380	if the player wants to go out, call the subroutine at line 1800.
1390	if the player wants to pray, call the subroutine at line 1870.
1400	if the player wants to go in, call the subroutine at line 1930.
1410	if the player wants to go north and this is possible, change the value of P%, otherwise print the message 'I can't do that!'.
1420	if the player wants to go south and this is possible, change the value of P%, otherwise print the message.
1430	if the player wants to go east and this is possible, change the value of P%, otherwise print the message.
1440	if the player wants to go west and this is possible, change the value of P%, otherwise print the message.
1450	if the player asks for the score, print the value of S%.
1460	if the player wants to 'get' an object, call the subroutine at line 2130.
1470	if the player wants to see the inventory of items they are carrying, call the subroutine at line 2300.
1480	if the player tries to 'drop', 'leave' or 'put' an object in the current location, call the appropriate subroutine.
1490	call the subroutine to wear an object.
1500	call the subroutine to pull an object.
1510	call the subroutine to wave the wand.
1520	call the subroutine to paddle across the river.
1530	call the subroutine to read the book.
1540	call the subroutine to 'say', 'talk' or 'repeat' the secret password.
1550	call the subroutine to 'attack', 'kill' or 'stab' an object.
1560	call the subroutine to search the current location.

1570	call the subroutine to throw an object being carried.
1580	call the subroutine to climb up.
1590	call the subroutine to go up.
1600	call the subroutine to drink.
1610	call the subroutine to give an object away.
1620	call the subroutine to 'use' or 'prise' an object.
1630	call the subroutine to swim across the river.
1640	call the subroutine to unlock the padlock.
1650	call the subroutine to oil the lock.
1660	call the subroutine to plant the beanstalk.
1670	call the subroutine to fill the vase.
1680	call the subroutine to pour the water.
1690	call the subroutine to go down.
1700	call the subroutine to 'press' an object.
1710	call the subroutine to ask for 'help'.
1720	call the subroutine to 'save' a game during play.
1730	call the subroutine to 'load' in a previously saved game.
1740	if the score is less than 10, start the loop again.
1760–1780	win the game. Print message, play a short tune and end the game.

In all games in this book and, for that matter, in all my adventures, the variable P% and S% are used to hold the player's current position and the score respectively. After setting the value of these variables in line 1080, the computer repeats the main loop until the score is greater than 9. You will notice that the game starts with the player standing outside the Wizard's cottage and therefore P% is set to 7 at the start of the game. Most games have a maximum score of either 10 or 100, but by changing the value tested for in line 1100, it is possible to write a game so that the player wins when any chosen score is reached. In some games, including the original *Colossal Caves,* the player is given some score at the start of the game and you may like to follow that example by changing line 1080.

The actual line numbers used in this game were changed many times during development of the program. When you try to write a program of your own, the reason for this will become quite obvious. After writing the standard section of code, the subroutines to deal with specific responses such as 'swim' or 'pray' were added one at a time to the program. Each time a new subroutine was added at the end of the program, a line was added between the line asking for your instructions and the end of the loop to call that routine. After a while, the space left between the lines began to run out and, in order to make the listing look neater, the program was renumbered. If you compare the main control section of this program with the others in this book, you'll see that they are essentially identical, although the line numbers will be very different.

The score is set to zero and the starting position to location number 7 in line 1080, whilst the main control loop from line 1100 to

1740 describes the location, the directions in which movement is possible and any objects visible before asking the player to type in their instructions. The LOWER$ instruction in line 1360 is used to convert the player's instructions into lower case. In this way, the player can type in either upper or lower case letters and the program will produce the same result. Remember, though, that the letters tested for in the following lines MUST be in lower case.

The variable Q%(P%) holds the description of location P% and this is printed in line 1110. You will notice that the score is calculated in a subroutine at line 2000 and this is called from the main loop every time round it. This routine could have been included within the main control loop, but using a subroutine meant that I was able to develop the program as a series of smaller modules, which made testing and debugging easier.

The section of code between lines 1140 and 1250 checks the directions in which players can travel and this information is held in the variable A$ so that it can be printed on the screen in line 1250. Some adventure game programmers prefer to include this information within the description of the locations by changing the DATA lines. To illustrate how to do this, consider location 7. The data holding its description is held in line 210 and this line could be changed to:

```
210 DATA outside a small cottage. A sign on the door reads 'Wizard
out at the moment. Please leave treasures inside'. I can go west or
into the cottage.,0,0,0,6
```

One major disadvantage of doing this is that you are then storing many more extra characters in the DATA lines and this uses far more memory than the method I have adopted, although you can leave out lines 1150 to 1250. The routine to sort out the directions for movement for north, south, east and west lies between lines 1150 and 1180, and this section of code is to be found in all three listings in this book. It first of all clears the contents of A$ and then checks whether the number held in S% (P%, 1) is greater than zero. If it is, then movement north is possible and therefore A$ is set to hold the word 'North'. In a similar way, the contents of S%(P%,2), S%(P%,3) and S%(P%,4) are checked to see if movement south, east or west is possible, and the contents of A$ are changed to include any possible directions.

The code between lines 1190 and 1230 then tests the number of the location to see if you can go up, down, in or out and again adjusts A$ if necessary. To illustrate this, consider location number 54, where you can go up if, and only if, the beanstalk has been planted in the fertile soil *and* you have watered it by filling the vase with water in the kitchen and then poured it over the plant. Unless the variable SL has been set to hold the number 1 (when the beanstalk has been watered), line 1220 will be ignored. A variable used in this way is called a flag, and you will find many such flags in adventure games. Once SL has been set to one, and you reach location 54, the variable A$ will tell you that you can now move up as well.

The final part of this section tests to see whether A$ is still empty, setting its contents to 'Nowhere obvious' if it is, before finally printing the directions of possible movement in line 1250. There are no locations in this game where the contents of A$ will still be empty, and line 1240 is not really necessary. It was included in this game just to illustrate how it can be done.

On reachng line 1280, the program tests the array B%(X) for each of the thirty objects found within the game to see whether the object is to be found in the current location (P%). In order to do this, the program uses the variables E and P as flags. P is set to hold the value 1 if any object is found in the current location, and unless this happens the program will not reach the section of code where the objects are described (lines 1320-1330). The variable E is set to one if more than one object is found in that position so that the message in line 1320 is not repeated for each object.

After the location, the directions and the objects have been described to the player, all that is left to be done is to input the player's commands and analyse them. This is done by comparing the first few letters of the player's instructions with the word which the programmer has decided will be relevant to the game, and if they match each other the appropriate subroutine is called.

Many advances have taken place in instruction decoding over recent years. The section of code from line 1360 to line 1730 is used to analyse the player's instructions but provides only the familar two word sentence decoding. It can be extended to provide more complex sentence decoding if you are prepared to spend some time developing it. (The subject of full sentence analysis is discussed in greater detail in Chapter 16.)

The method I've adopted for decoding the player's instructions is to store the first two letters in B$, the first three letters in C$ and first four letters in D$ in line 1360. The following line doesn't play any part in the decoding and may be left out if you don't mind the screen scrolling during play. The remaining lines in this control section are used to compare these variables (B$, C$ and D$) with set word patterns corresponding to the instructions which you want the computer to recognise and, apart from the sequence to move north, south, east or west, pass control to an appropriate subroutine. The only difficulty with using this method is that the computer will then match a number of words with the routine. Consider the player who tries to thrash the vampire. This will be interpreted as 'throw', which will result in a totally unexpected response. You may like to change the routine so that it compares the full word, but this too can have its disadvantages.

Rather than type in the rest of the control section in one sitting, it is easier to debug the program if you type in one line at at time and then develop its associated subroutine. You can then check that the subroutine works correctly before moving on to the next line. As an example of this, suppose you want to add an extra subroutine so that the computer recognises and understands sentences beginning with the word 'dive'. The following line should be added to the control section.

```
1731 IF C$="div" THEN GOSUB 10000
```

You'll notice that I've added the subroutine at a line number well past the end of the main program. This is to allow plenty of space for the routine. Once you are convinced that it works correctly, the program can be renumbered to make it easier to follow and also allow space for the next subroutine. The only part of the main control sequence which acts upon the player's instructions without using a separate subroutine is that dealing with movement in the prime compass directions (north, south, east and west). The code needed to control this movement is so simple that it isn't worth writing it in a subroutine. Line 1410 deals with movement to the north. The first part of the line checks whether the player has typed an instruction beginning with 'go n' or simply 'n'. If this is the case, the computer then checks the contents of the array elements S%(P%,1), which holds the map for movement north from the current location. Should this element contain a number greater than zero, then this number will represent the number of the location reached by travelling north from the current location, and the value of P%, the current location, is changed to this new number. The next three lines deal with movement south, east and west by examining the contents of S%(P%,2), S%(P%,3) and S%(P%,4) in a similar manner.

Once the computer reaches the end of the loop, the score (S%) is tested and if it is still less than ten, the computer jumps back to the beginning of this loop and starts the process all over again. Careful study of this loop shows that the computer doesn't print any message if it fails to recognise the player's instructions. This is an easy, and extremely useful, feature to add to any adventure game and can make the program have a much more 'human' quality, especially if the responses are humorous. To do this, we can use another flag and set its value to zero immediately after the player's instructions are input. If the command given by the player is then recognised by the computer, the value of this flag should be changed to a value other than zero, so that an extra line can be added to test its value at the end of the loop and a message can be printed if the flag is still zero. For example:

```
1370 PRINT CHR$(7):CLS:K=0

1380 IF C$="out" OR D$="go o" THEN GOSUB 1800:K=1
```

```
1735 IF K=0 AND LEN(Z$)›0 THEN PRINT" I'm sorry I don't seem
to understand your instructions. Perhaps you should rephrase you
command."
```

In the above example, the variable K would be set to one at the end of each line where an instruction was understood and would remain zero only if the word pattern were not recognised. In line 1735, the message would be printed if the value of the flag were still zero and the player had typed an instruction rather than just pressing ‹ENTER›. Until all of the subroutines have been typed in, you will be unable to test that the main control section works fully, but should be in a position to check that you can move around the adventure. Before proceeding with the next chapter, it's worth spending a little time checking that you can move north, south, east and west in your game. The easiest way of doing this is to RUN the program and when the computer prints the message asking for your response, escape from the game by pressing the ESC key twice. You can than change your location, without actually playing the game, by typing P%=1 and pressing ‹ENTER›. This will move you to location 1. You can then continue the game by typing CONT and pressing ‹ENTER›. In this way, you can move to each of the eighty locations in the game and test whether the movement routine works as you expect. If all is well, don't forget to save a copy of your game. Should a fault occur at this stage, however, you will need to check the numbers in the data lines to ensure that you have typed them in correctly; also check the lines 1410 to 1440 for typing errors.

Setting the puzzles: part 1 4

Now that we have completed the routine part of the game, we can really get to grips with the most interesting part of the whole process. Setting puzzles and problems for the player to pit his wits against is a very time consuming procedure and needs to be tackled in several stages. Some of the subroutines called by the main control section will be common to all adventure games. Examples which readily spring to mind are those dealing with handling objects, such as 'get', 'drop' and 'inventory', whilst others deal with your position in the game such as 'score', 'help' and 'look'. Some routines, however, will be unique to the particular game and although many adventurers argue the fun of playing the game comes from finding out which words the computer recognises, you would be well advised to give the player some information about the words which are understood by the computer. As the level of decoding by the computer becomes more complex, it becomes ever more important to give the player this information to point him in the right direction and so avoid a great deal of needless frustration. This may be achieved by providing a printed instruction sheet to accompany the program or by including instructions with the game. In *The Wizard's Quest,* there are two subroutines which are called not from the main control section but from other subroutines. They are, in fact, the two most important subroutines in the whole game and, for that reason, will be discussed first.

Losing the game

The first of these, from line 2600-2650, deals with losing the game. In any adventure, there will be many occasions where the player loses his life by performing foolhardy tricks such as jumping from the roof of a burning building or swimming in crocodile infested water; writing separate routines to deal with every possible death would be very wasteful of memory. Before calling this routine, a message describing the death *must* be stored in the variable E$, which is then printed at the top of the screen. The player will then have to press the space bar for another game.

Subroutine for losing the game

```
2600 REM ** lose game **
2610 CLS:PRINT E$:LOCATE 1,20:PRINT"Press the <Spa
ce Bar> for another game."
2620 WHILE A$<>" "
2630 A$=INKEY$
2640 WEND
2650 RUN
```

Line

2610	clear the screen, print the message held in E$ and print the message about pressing the space bar.
2620	wait for the space bar to be pressed.

Splitting sentence

The second routine, from line 2230 to line 2280, is probably the most important in the whole game. Its first purpose is to split the instruction typed in by the player into two separate words, and store the second word of the sentence in the variable L$. The first few letters of the input sentence have already been stored in B$, C$ and D$, and the computer will have already recognised the player's intention in principle, although not in detail! Supposing, for example, that the player types the instruction 'get rope', then the main control section would call the subroutine which deals with 'get' and this, in turn, would call the routine being discussed to find out which object the player wants to get and hence, on returning to the 'get' routine, the variable L$ would hold the word 'rope'. In order to do this, the computer uses the INSTR command to search the input string (Z$) for the first occurrence of a blank space (" ") and sets L$ to hold that part of Z$ to the right of it. Obviously, if the player types in 'get the rope', L$ would then hold the words 'the rope', which would not be recognised in the following lines. Games which include full sentence decoding would then have to search L$ for any other occurrences of a blank space and leave the final value of L$ with just the word 'rope' in it. You may like to try experimenting with more complex sentence analysis for yourself when you have sorted out the main sequence.

Subroutine to split sentence

```
2230 REM ** check item **
2240 L$="":XX=INSTR(Z$," "):R=0
2250 L%=0:L$=RIGHT$(Z$,(LEN(Z$)-XX))
2260 IF LEN(L$)<2 THEN RETURN
2270 FOR X=1 TO 30:IF LEFT$(N$(X),LEN(L$))=L$ THEN
 L%=1:R=X
2280 NEXT:RETURN
```

Lines 2270-2280 then search through the contents of the 30 elements of the array N$(X), which holds the names of the objects recognised by the computer, to see if the contents of L$ match any of the known objects in the game. If a match does occur, then the variable R is set to hold the number of that object. Should the object not be recognised, then the value of R will remain zero when control is returned to the subroutine which called it. In *The 'Wizard's Quest,* for example, the rope is not found until the player has searched the leaves for it and thus if the player tries to 'get rope' before searching the leaves, the value of R returned would be zero. After searching in the right places, however, the variables N$(11) and G$(11) are changed and trying to 'get rope' would then return a value of R=11.

Line	
2240	empty the contents of L$, find the position of the blank space in the string and set the value of R to zero.
2250	set the flag to zero and change L$ so that it holds the second word typed in by the player.
2260	check the length of the word held in L$ and if it is too short, return to the calling subroutine.
2270	search through all 30 objects to see if the word held in L$ matches the description of any of them and set R to the number of the object if it does.
2280	return to the calling subroutine.

Calculating the score

All other routines, with the exception of that used to calculate the player's score, are called as a direct result of the player's instruction. The subroutine used to calculate the score, however, is called every time round the main control loop, so that the computer always has an up to date score to check at the end of the main section. There are many different ways of giving the player a score in an adventure game, and the routine from line 1990 to 2110 illustrates one of the most popular methods. If you can remember back that far, the ten items of treasure to be found have to be taken and dropped inside the Wizard's cottage, location 12. The ten items of treasure discussed in Chapter 1 were object numbers 13, 15, 17, 18, 19, 20, 21, 28, 29 and 30. Each time the routine is called, the score is set to zero in line 2000 to make sure that it doesn't build up on its previous value without the player finding any further items of treasure. Lines 2010 to 2100 then check the location of the treasures to see whether they are inside the cottage and increase the score by one for each of the above objects found.

Subroutine to calculate the score

```
1990 REM ** set score **
2000 S%=0
2010 IF B%(13)=12 THEN S%=S%+1
2020 IF B%(15)=12 THEN S%=S%+1
2030 IF B%(17)=12 THEN S%=S%+1
2040 IF B%(18)=12 THEN S%=S%+1
2050 IF B%(19)=12 THEN S%=S%+1
2060 IF B%(20)=12 THEN S%=S%+1
2070 IF B%(21)=12 THEN S%=S%+1
2080 IF B%(28)=12 THEN S%=S%+1
2090 IF B%(29)=12 THEN S%=S%+1
2100 IF B%(30)=12 THEN S%=S%+1
2110 RETURN
```

Line

2000	set the score to 0.
2010	if object number 13 is in location 12, increase the score by 1.
2020–2100	repeat this process for the other treasures.
2110	return to the main program loop.

'Get', 'inventory' and 'drop'

All adventure games need routines which allow the player to pick up and drop objects and the next three subroutines deal with this topic. The subroutine which allows the player to 'get' an object is called from line 1460 in the main control loop. In order to make the game as 'user friendly' as possible, the computer also recognises the words 'take' and 'grab'.

The 'get' routine

```
2120 REM ** get routine **
2130 GOSUB 2240:IF L%<1 THEN RETURN
2140 E%=0:FOR X=1 TO 30:IF B%(X)=P% AND N%(R)=X TH
EN E%=1
2150 NEXT:IF E%=0 THEN RETURN
2160 IF (R=13 AND SI=0) OR (R=15 AND SJ=0) OR (R=2
1 AND SN=0) OR R=26 THEN PRINT"Don't be absurd!":R
ETURN
2170 IF R=12 THEN PRINT"I can't carry them all!":R
ETURN
2180 A(R)=1
2190 E%=0:FOR X=1 TO 4
2200 IF V$(X)="" THEN V$(X)=G$(N%(R)):E%=1:X=5
2210 NEXT:IF E%=0 THEN PRINT"Sorry. My hands are f
ull!":RETURN
2220 B%(N%(R))=0:RETURN
```

The first thing that this section of code does is to call the subroutine discussed previously to split the sentence into two words. When control is returned to the 'get' routine, the program tests the flag L% to see whether the second word typed in has been recognised. If L% still contains zero, then the player has typed in the name of an object which the computer doesn't recognise and the program returns to the main loop without any comment. One suggestion which you may like to try out would be to insert a message into this line before returning to the main loop:

```
2130 GOSUB 2240: IF L%<1 THEN PRINT "I can't see a ";L$;" here
!":RETURN
```

Most adventure game players appreciate a little humour, so try to introduce a little wit into your comments!

Line	
2130	call the subroutine to analyse the sentence and if the flag is zero on return, return control to the main loop of the program.
2140	check all thirty objects to see if they match the object mentioned by the player and if they are in the current location, set the value of the flag E% to one.
2150	if E% is zero when this line is reached, return to the main loop.
2160	check whether the object can be picked up. If it is not possible, print the message and return to the main loop.
2170	prevents the player from carrying object number 12.
2180	set the value of the flag for the object so that the computer knows that it is being carried.
2190–2200	insert the description of the object into the array V$(X) which holds the inventory of objects being carried.
2210	if the value of the flag is zero, print the message and return to the main loop.
2220	remove the object from the current location and return to the main loop.

If the object that the player wants to get has been recognised by the computer then L% will equal 1 and the computer will then check through all thirty elements of the array B%(X) to see whether the object mentioned by the player is in the current location, P%. This is done in lines 2140 to 2150, where the variable E% is used as a flag to check that object number (R) is to be found in location P%. Should the value of E% still remain zero at the end of line 2150, then the object is not to be found in the current location and control is returned to the main loop. I have again included no message to tell the player that the object is not there and you may like to change line 2150 to something like:

```
2150 NEXT: IF E%=0 THEN PRINT "Maybe I need glasses, but I just
don't see ";L$;"here !":RETURN
```

When developing this game I decided to illustrate as many different methods of setting problems in adventures as I possibly could. Line 2160 is an example where the player is prevented from carrying objects numbers 13, 15 or 21 unless the variables SI, SJ or SN have been set to 1. If you can remember back to Chapter 1, these are three objects which start out as 'monsters', but which change into treasures later.

Object number	*Starts as*	*Finishes as*
13	A vampire	A jade ring
15	A giant slug	A silk purse
21	A giant	An emerald

The variables SI, SJ and SN are used as flags to test whether the player has got rid of the 'monster' and found the treasure. Also in line 2160, you will see that I have prevented you from carrying object number 26, an angry farmer and in a similar way, you are prevented from carrying the leaves in line 2170.

Whilst on the subject of variables used as *flags*, I should like to mention that I have used the variables from SA through to SP as flags to test whether a problem has been solved. I would strongly recommend that each time you introduce a new variable as a flag you also make a note of it on a piece of paper. The main reason for this is that when you come to write a routine to save your position onto tape or disc, you must ensure that the values of all the flags used in this way are saved alongside the other variables. To help me keep track of the flags, I tend to write down the names of the variables on paper before actually using them, and in this game I decided on the series SA to SZ. Each time that a new variable is introduced into the game, I would then tick it off the list. Later on in the game, we will come across instances where the computer needs to know whether the player is carrying a particular item. One example of this is where the player must kill the wolf before being able to progress further into the game. This routine was written in such a way that the player dies whilst attempting to do so unless he is carrying the sword. It is very important, therefore, that the computer knows which items are being carried at any instant and this is achieved in line 2180. Thus if the player types 'get oil', the value of A(2) would be set to 1 because object 2 is the can of oil!

In this game, the player is allowed to carry only four items at any one time, and therefore lines 2190 to 2210 check the four elements of the array V$(X) to see if they are empty. If all four elements are full, then the value of E% remains zero and an appropriate message is given in line 2210 before returning to the main program loop. Should an empty location in the array be found, then its contents will be changed in line 2200 to hold the description of the object and the value of X increased so that the loop is terminated. If X were not increased to 5, then all the elements of the array following the empty one would hold the same object! Finally, line 2200 sets the pointer B%(X) so that the object no longer appears in any location in the game. On returning to the main loop, the object appears in location zero, which doesn't exist!

The 'inventory' routine

```
2290 REM ** inventory **
2300 E=0:PEN 1:PRINT"I am carrying :-":PEN 2
2310 FOR X=1 TO 4:IF V$(X)<>"" THEN PRINT V$(X):E=
1
2320 NEXT:IF E=0 THEN PRINT"Nothing at all!"
2330 IF A(6)=2 THEN PRINT"I am wearing the gloves!
"
2340 IF A(5)=2 THEN PRINT"The vase is full of wate
r"
2350 RETURN
```

When a player wants to find out what they are carrying, they would normally type in 'inventory' and in this game, the routine to deal with it can be found from line 2290 to 2350. This routine is very simple and needs little explanation other than to discuss the tests in lines 2330 and 2340. These tests check whether the player is wearing the rubber gloves and whether they have filled the vase with water. The contents of the array A(R) will normally be zero if object number R is not being carried, or 1 if it is. In the routines we will come across later, the values of A(6) and A(5) are set to 2 if the player wears the gloves or fills the vase. Thus if they drop the gloves, A(6) will be set to zero again, indicating that they have been removed first and if the player drops the vase, the water will spill out and A(5) will also be set to zero.

Line	
2300	set the value of the flag to zero and print the message.
2310	search through all four elements of the array V$(X) and if it contains something, print the description of the object and set the value of the flag to 1.
2320	if the value of the flag is still zero, print the message that nothing is being carried.

2330 check the flag to see if the player is wearing the gloves and print the message if they are.
2340 check the flag to see if the player has filled the vase with water and print the description if they have.
2350 return to the main program loop.

The 'drop' routine

```
2360 REM ** drop item **
2370 GOSUB 2240:IF L%<1 THEN PRINT"I don't have ";
L$:RETURN
2380 E%=0:FOR X=1 TO 4
2390 IF V$(X)=G$(N%(R)) THEN V$(X)="":E%=1
2400 NEXT:IF E%=0 THEN PRINT"I'm not carrying ";L$
:RETURN
2410 B%(N%(R))=P%
2420 A(R)=0
2430 IF R=25 AND P%=72 THEN PRINT"The farmer smile
s and thanks me. 'I've  searched all day for him.
Please take mysling. You'll find it useful!', he s
ays.":G$(27)="a sling":G$(25)="":G$(26)="":B%(26)=
0:B%(25)=0
2440 RETURN
```

Now that the player is able to pick up objects found in the game and the computer is able to tell them which items they are holding, the next stage of development is to allow the player to drop objects being carried. Just as the main control loop recognised 'take' and 'grab' as alternatives to 'get' when calling that subroutine, the computer has also been instructed in line 1480 to recognise 'leave' or 'put' as alternatives to 'drop' when calling the subroutine between lines 2360 and 2440.

The first line of this subroutine again calls the subroutine at line 2240 whch splits the input sentence into two words. If the object that the player tries to drop is not recognised by the computer, then the value of L% will remain zero and the message in line 2370 will be printed before returning to the main loop. Lines 2380 to 2400 search through all four elements of the array V$(X), which holds the items being carried, to check whether the player is, in fact, carrying the item in question. Should the array elements not contain the object, then control is returned to the main loop after printing the appropriate message in line 2400.

Line 2410 is used to set the pointer B%(N%(R)), which tells the computer which location the object is to be found in, back to the current location, P%. The next line then sets the contents of the array A(R) back to zero so that the computer knows that the player isn't carrying the object any longer.

Before returning to the main control loop, line 2430 checks whether the player has dropped the dog in location 72, where the farmer stands looking for him. When this is done, one of the invisible objects, number 27, is changed into the sling, which will be needed later in the game, and the pointer B%(X) is set to zero for objects 25 and 26. These two objects are the dog and the farmer respectively and this has the effect of moving them to location zero, which doesn't exist, to give the appearance of them moving away. Many puzzles in adventure games can be set in this way and it's worth while examining this line very carefully to make sure that you understand how it works.

Line

2370	calls the subroutine to split the player's sentence into two separate words. If the value of the flag L% is zero on return, print the message that it isn't being carried and return to the main loop.
2380	set the flag to zero and search the four elements of the array V$(X) which holds the inventory of items being carried.
2390	if the word asked for is the same as an object being carried, empty that element of the array V$(X) and set the value of the flag to 1.
2400	if the value of the flag is still zero, then the object is not being carried and therefore the message is printed before returning to the main control loop.
2410	set the pointer for the position of the object to the current location (P%).
2420	set the value of the flag for the object to zero so that the computer knows that it is no longer being carried.
2430	check whether the player has dropped the dog at the farmer's feet and solve the puzzle if they have.
2440	return to the main program control loop.

Testing

Once you have reached the point where you have typed all the routines in this chapter into your computer, you would be well advised to test that they work properly before proceeding further. The first thing to check is that the 'get', 'inventory' and 'drop' routines work correctly and the easiest way of doing this is to RUN the program and try 'getting' any objects you find. Check that the inventory routine works when you are carrying no objects and then when you are carrying four objects. Also check that the computer will not permit you to pick up more than four objects at any one time and that you can 'drop' them again.

In addition to checking that the general routines work, you will also need to check that you can't pick up the leaves, object 12, which are found in location 24. The easiest way to test this is to escape from the program by pressing the escape key twice, typing P%=24, to change location, and then typing CONT before pressing ‹ENTER›. You should then try to get the leaves. Any mistake at this point should be fairly easy to track down by a process of elimination, but if you leave debugging until the end, you will find life much more difficult.

Checking that you can't 'get' the vampire, the slug or the giant can be achieved in a similar way. You will need first of all to escape from the program and change the value of P% to the location you wish to visit before CONTinuing with the program. If you want to test that you can get these objects after solving the problems, you will need to wait until you have read the next chapter.

The program should also be tested to see whether dropping the dog in location number 72 produces the right effect. Again you should escape from the program and move to location 54, where the dog is to be found and CONTinue with the program. Once you have got the dog, you should once more escape from the program and more to location 72, where you can, after CONTinuing, drop the dog. At this stage, you will only be able to test that the 'lose game' routine works by escaping from the program and typing GOSUB 2610 ‹ENTER›. This doesn't test the routine out fully because the only sure test that it works is by attempting to kill yourself, but these sections have not yet been written! For similar reasons, we will be unable to check the routine which works out the score, except by careful comparison with the listing. One final check that can be made is that typing 'help' produces the reply 'I'm sorry I don't have a clue!'. Once you are sure that all is well, don't forget to save a copy.

Setting the puzzles : part 2 5

Unlike the routines described in the previous chapter, which are essential in all adventure games, many of the routines here were developed specifically for this game. In the process of developing them, you should find many ideas which can be adapted and used in other games.

GO OUT

This command is useful when the player finds himself inside a room and the instruction is recognised in line 1380 when the player types 'go out' or simply 'out'. There are five locations in *The Wizard's Quest* where this command is useful and these are summarised below:

Location	*Description*	*Leads to location*
12	Inside cottage	7 outside cottage
22	Inside chapel	21 outside chapel
25	Living room	29 outside cottage
63	Inside statue	64 outside statue
70	Inside barn	69 outside barn

```
1790 REM ** go out **
1800 IF P%=12 THEN P%=7:PRINT Y$:RETURN
1810 IF P%=22 AND SA=0 THEN PRINT"The door's locke
d!":RETURN ELSE IF P%=22 THEN P%=21:PRINT Y$:RETUR
N
1820 IF P%=25 THEN P%=29:PRINT Y$:RETURN
1830 IF P%=63 THEN P%=64:PRINT Y$:RETURN
1840 IF P%=70 THEN P%=69:PRINT Y$:RETURN
1850 PRINT"Don't be silly!":RETURN
```

The variable Y$ was set at the beginning of the program to contain the most useful message in the whole game, 'O.K.'. In each of the lines between 1800 and 1840, the computer checks to see whether the location is one of those listed above. If the current value of P% does correspond to a location where the player can go out, then the message 'O.K.' is printed, the value of P% is changed to the value shown in the chart and control is returned to the main section of the program.

The only location where this is not quite true is inside the chapel. When the player first enters the chapel, the door slams shut behind him and unless he tries 'praying', the door will stay firmly locked! This little puzzle is an easy one to set for the player. The variable SA is used as a flag to test whether the player has prayed inside the chapel. Unlike a number of machines, variables don't need to have their value set to zero at the start of the program in Locomotive BASIC. This means that SA will start at zero and in the 'pray' routine this will be set to 1. In line 1810, the computer will print an appropriate message if the door is locked, but will allow the player to move outside if the door is open. If the program reaches line 1850, then it will have checked all the locations where the player can go out and will print a suitable message to indicate that the player is being stupid.

Line	
1800	if in location 12, change the value of P% to 7 and return to the main loop.
1810	if in location number 22 and the door is locked, print an appropriate message and return to the main loop without changing the value of P%, otherwise, change the value of P% to 21, print the message and return.
1820	if in location 25, move to location 29 by changing the value of P% and return to the main loop.
1830	if in location 63, change the value of P% to 64, print the message and return.
1840	if in location 70, move to location 69, print the message and return.
1850	print the message that it isn't possible to go out and return to the main loop of the program.

Testing
It's always easier to test that a routine works when you've just typed it in and the ideas are fresh in your mind. The first thing to do when testing this routine is to RUN the program and escape from it by pressing the escape key twice. Provided that you don't attempt to edit any line of the program, all the variables will remain intact and you can change the value of the current location by typing P%=12

and pressing ‹RETURN›. Try out the routine by typing 'go out' to check that you do in fact end up in location 7. Test the other locations in a similar way. When you are in the chapel, however, you will be told that the door is closed, and to test the routine fully, you will have to wait until you've typed in the 'pray' routine.

PRAY

This routine is called from line 1390 in the main game when the player types 'pray'.

```
1860 REM ** pray **
1870 PEN 2:PRINT Y$
1880 ENV 1,100,2,2:ENT 1,100,-2,2:SOUND 1,284,200,
1,1,1
1890 IF P%<22 OR P%>23 THEN PRINT"That made me fee
l better!":RETURN
1900 IF SA=0 THEN PRINT"The door opened !":SA=1:Q$
(22)=LEFT$(Q$(22),24):RETURN
1910 PRINT"The door closed !":Q$(22)=Q$(22)+" The
door has   closed behind me!":SA=0:RETURN
```

The computer first prints up the message 'O.K.' in line 1870 and then plays a short tune in line 1880 before checking whether the player is inside the chapel. Praying outside the chapel has no effect and control is returned to the main loop in line 1890. Line 1900 then checks to see that the door is closed, SA=0, and changes the value of SA to 1 if it is. It also shortens the description of location 22 to remove the message that 'the door has closed behind me'. If the door is already open, then the flag SA is set back to zero and the door closed again (line 1910).

Line	
1870	print the message set at the beginning of the program.
1880	sound effects to accompany prayer ... adjust these to suit your own requirements.
1890	if not inside the chapel, locations 22 and 23, print the message and return to the main program control loop.
1900	check the value of the flag and open the door if it is closed, by changing the value of the flag SA, before returning to the main loop.
1910	if this line is reached, the door must be open, so the value of the flag SA is changed to zero to close it and control is returned to the main loop again.

GO IN

```
1920 REM ** go in **
1930 IF P%=7 THEN P%=12:PRINT Y$:RETURN
1940 IF P%=21 AND SB=0 THEN PRINT"The door's locke
d!":RETURN ELSE IF P%=21 THEN P%=22:PRINT Y$:RETUR
N
1950 IF P%=29 THEN P%=25:PRINT Y$:RETURN
1960 IF P%=64 AND SC=0 THEN PRINT"The way in is cl
osed!":RETURN ELSE IF P%=64 THEN PRINT Y$:P%=63:RE
TURN
1970 IF P%=69 AND SH=1 THEN P%=70:PRINT Y$:RETURN
ELSE IF P%=69 THEN PRINT"He won't let me!":RETURN
1980 PRINT"Don't be stupid!":RETURN
```

This is one of the more frequently used commands in this game and is called from the main program loop by typing 'go in' or simply 'in'. Like the routine to 'go out', there are five locations where you can use this instruction to effect.

Location	*Description*	*Leads to location*
7	Outside cottage	12 inside cottage
21	Outside chapel	22 inside chapel
29	Outside cottage	25 living room
64	Outside statue	63 inside statue
69	Outside barn	70 inside barn

Line 1930 checks to see if the player is in location 7 and changes the value of P% to 12 if he is. In a similar way, line 1950 checks whether the player is in location 29 and changes the value of P% to 25.

Movement in the other three locations is not as easy for the player because they have to solve certain problems first. In location 21, they cannot go into the chapel unless the door is opened by pulling the lever. This, in turn, will result in death unless they are wearing the rubber gloves to prevent them from getting an electric shock! Once the lever has been pulled, the value of SB is changed to one and when the player tries to enter the chapel the value of SB is tested in line 1940 to see whether the door is open or not. In a similar way, the player will not be able to enter the statue until they have pressed the correct button (SC=1) and in addition, the tramp will not permit entry into the barn until he has been given the bottle of rum (SH=1).

Should the program reach line 1980, then the player must have tried to go into a location other than the five listed above and a message to tell them that they can't do this is printed. Testing of this subroutine should be carried out in a similar way to the 'go out' routine, although you won't be able to check the final three locations fully until you have entered the appropriate routines where the variables SB, SC and SH are set to 1. If you do want to test that you can go into these locations, try changing the value of these variables by escaping from the program, changing their value and then typing CONT.

Line	
1930	if in location 7, move to location 12, print the message and return to the main loop.
1940	if in location 21 and the flag is zero, print the message about the door being closed and return to the main loop, otherwise change the value of P% to move to location 22 and return to the main loop.
1950	if in location 29, move to location 25, print the message and return to the main loop.
1960	if in location 64, and the flag is zero, print an appropriate message and return to the main control loop, otherwise move to location 63 and return to the main loop of the program.
1970	if in location 69, and the flag is 1, move to location 70, otherwise print the message about the tramp not allowing the player to go in and return to the main loop without changing the value of P%.
1980	print the message about movement not being possible and return to the main control loop.

WEAR

```
2450 REM ** wear **
2460 GOSUB 2240
2470 IF R<>6 THEN PRINT"Don't be silly!":RETURN
2480 IF A(6)=0 THEN PRINT"I don't have them!":RETU
RN
2490 IF A(6)=2 THEN PRINT"I'm already wearing them
!":RETURN
2500 A(6)=2:PRINT Y$:RETURN
```

When the player tries to pull the lever outside the chapel, they will get an electric shock and die unless they are wearing rubber gloves. If you check through the list of objects, you will see that object 6 is 'a pair of rubber gloves'. The first stage in checking whether the player intends to try wearing anything else found in the game is to call the 'check routine', which has already been discussed. If the player does type in 'wear gloves', then the value of R will be set to 6 and line 2470 will not then return to the main loop. Attempting to wear anything else is not possible. Line 2480 then checks the value of A(6) to make sure that the player is carrying the gloves and in line 2490 a check is made to ensure that the player is not already wearing them. If the player is in a position to wear them, then the value of A(6) is set to 2 in line 2500. The value of A(6) is later checked in the routine to 'pull' and in the 'inventory' routine.

Line	
2460	call the subroutine to split the sentence into two separate words.
2470	if the object is not number 6, the gloves, print the message and return to the main control loop.
2480	if the flag is zero, print the message and return to the main loop.
2490	if the value of the flag is 2, print the message about already wearing them and return to the main loop.
2500	set the value of the flag to 2, print the message and return to the main loop.

PULL

```
2510 REM ** pull lever **
2520 GOSUB 2240:IF P%<>21 OR LEFT$(L$,3)<>"lev" TH
EN PRINT"Don't be silly!":RETURN
2530 IF A(6)<>2 THEN E$="A violent electrical curr
ent surges      through my body. I am dead!":GOSUB
2610
2540 PRINT"The door opens!":SB=1:RETURN
```

The only way into the chapel, location 21, is by pulling the lever. Line 2520 checks that the player is in location 21 and that they don't want to pull anything else. In this game, the only object which can be pulled is the lever and this is not an 'object' which is included in the list of objects recognised by the computer. It is, in fact, mentioned only in the description of the location (Q$(21)).

The next check, in line 2530, is that the rubber gloves are worn. Should A(6) not equal 2 then the player receives a violent electrical shock and control is passed to the death routine (line 2610). If the player is wearing the gloves, then the value of SB is set to one in line 2540 and this allows the player to go into the chapel (see 'go in' routine).

Once this routine has been typed in, you are in a position to check that all the puzzles connected with the chapel work. The easiest way to move around is to escape from the program and change the value of P% before CONTinuing with the program. Go to the cottage and find the gloves. Take these back to the chapel and try pulling the lever. Go into the chapel and try praying. If all is well save your updated copy on tape or disc. Any faults should be checked carefully against the listing.

Line

2520	call the subroutine to split the player's instructions into two words. if the second word is not the lever, print the message and return to the main loop.
2530	if the player is not wearing the rubber gloves, the flag A(6) will not be 2 and the player will die.
2540	print the message, set the value of the flag for the door to open and return to the main control loop.

WAVE

```
2550 REM ** wave **
2560 GOSUB 2240:IF R<>7 THEN PRINT"Don't be stupid
!":RETURN
2570 IF A(7)=0 THEN PRINT"I don't have it!":RETURN
2580 IF P%<>42 OR S%(42,2)>0 THEN PRINT"nothing ha
ppens!":RETURN
2590 PRINT"The drawbridge comes down!":S%(42,2)=46
:RETURN
```

One of the standard features of many traditional adventures is the magic wand. In this game, the player must find the wand and wave it at the side of the bottomless pit, location 42. Waving any other object doesn't do anything and this is checked for in line 2560. Line 2570 checks that the player is carrying the wand, object 7, and returns control back to the main routine if A(7) remains zero. The location and the value held in S%(42,2) is then checked in line 2580 to see if the player is in the correct place and whether he can go south. Control is returned to the main section if the player is in any location other than 42 or if the bridge has already come down. Line 2590 changes the value held in S%(42,2) so that movement south now takes the player to location number 46 and the message that the drawbridge comes down is printed on the screen.

Testing this routine involves going to find the wand and taking it back to the correct location before waving it. The quickest way of moving from one place to another is again by escaping from the program and changing the value of P%.

Line

2560	call the subroutine to split the sentence into two words and if the second word is not number 7, print the message and return.

2570 check to see if object 7 is being carried, print the message and return.
2580 if not in position 42 or if the map has been changed, print message and return.
2590 print the message, change the map and return.

READ

```
2660 REM ** read **
2670 IF A(9)=0 THEN PRINT"I have nothing to read!"
:RETURN
2680 PRINT"The book makes interesting reading.
 'To enter the caverns, repeat the runes  SDFDA'"
2690 A(0)=1:RETURN
```

The only object which can be read by the player in this game is the book of spells, object 9. Line 2670 checks that the book is in fact being carried and returns control to the main control loop if it isn't. Line 2680 then gives the player the information needed to get into the caves. It also sets the value of A(0) to 1 which acts as another flag telling the computer that the player has read the message. This means that the player has to read the book each time he plays the game — memorising the runes will not work! You may like to try changing the code needed to enter the caves to a random sequence of letters to make the solution harder to find!

Once again, this routine cannot be fully tested until later, when you have entered the routine which allows you to speak.

Line

2670 check to see if the player is carrying anything to read and print the message before returning to the main loop if he isn't.
2680 describe how to enter the caves.
2690 set the value of the flag and return to the main control loop.

TALK

```
2700 REM ** talk **
2710 CLS:INPUT"What do you want to say ";Z$:Z$=LOW
ER$(Z$)
2720 IF P%<>27 THEN PRINT"I talk but nobody listen
s!":RETURN
2730 IF Z$<>"sdfda" THEN PRINT"Nothing happens":RE
TURN
2740 IF A(0)=0 THEN PRINT"It didn't work!":RETURN
2750 Q$(27)="at the entrance to a large cavern."
2760 A(0)=0:S%(27,2)=31:PRINT"The caverns open!":R
ETURN
```

The only way into the caves, location number 27, is by saying the correct password. This routine is called from the main section by the instructions 'say', 'talk' or 'repeat' and the first line of the routine asks the player to INPUT the word they want to say. If the location is not correct or if the player types in the wrong code, control is passed back to the main loop of the program. Line 2740 is used to check that the player has read the password during *this* game and has not tried to remember it from a previous game. This line is optional and may be left out if you don't mind the player remembering the password.

Line 2750 changes the description of location number 27 and line 2760 then changes the map to allow movement south from the caves.

If you have already typed in the 'read' subroutine, then you are in a position to test out the puzzle of entering the caves. As in all testing, you should escape from the program and change the value of P% so that you move to the correct locations. Get the book, take it to the cave entrance and check that reading the book and saying the password does, in fact, open up the cave entrance.

Line	
2710	input the word that the player wants to say.
2720	if the location is incorrect, print the message and return to the main loop.
2730	if the first few letters of the word are incorrect, print the message and return to the main loop.
2740	check the value of the flag to see if the player has read the password. If he hasn't, print the message and return to the main loop.
2750	change the description of the location.
2760	set the value of the flag, change the map, print the message and return to the main loop.

KILL

```
2770 REM ** kill **
2780 GOSUB 2240:IF R=13 OR R=15 OR R=21 OR R=25 OR
 R=26  THEN PRINT"That's not the right approach!":
RETURN
2790 IF LEFT$(L$,3)<>"wol" OR P%<>20  THEN PRINT"D
on't be absurd!":RETURN
2800 IF A(10)=0 THEN E$="The wolf attacks me!!":GO
SUB 2610
2810 PRINT"The wolf dies!!":S%(20,4)=24:Q$(20)=LEF
T$(Q$(20),17):RETURN
```

In this game, the player must kill the wolf, which is found in location 20, before he can move west from there. The wolf is not one of the 'objects' found in the game and is, like the lever described in the 'pull' routine, only mentioned in the description of the location. Line 2780 checks which object the player wants to kill and suggests that he is not adopting the right approach if he attempts to kill the vampire, the slug, the giant, the dog or the farmer. The following line checks that L$ contains the word 'Wolf'. This is necessary because the wolf is not a true 'object' with its own value of R. There is also a check that the player is in the correct location to kill the wolf. If the player is not carrying the sword, then the wolf attacks and the 'lose game' routine is called (line 2800). If the program reaches line 2810, then the player must be in location 20 and must be carrying a sword with which to kill the wolf. This line prints the message that the wolf dies, changes the map so that the player can move west from location 20 and changes the description of the location to eliminate any mention of the wolf. Thus the player can now move west to location 24.

Testing this routine is quite straightforward and should be completed before proceeding with the next routine.

Line

2780	call the subroutine to split the sentence into two words and if the object mentioned is too dangerous to kill, the message is printed before control is returned to the main loop.
2790	if the second word typed in by the player is not 'wolf', the message is printed and control returned to the main loop
2800	if the flag is zero, the player isn't carrying the sword, so the message is stored in E$ before the death routine is called.
2810	print the message, change the map and the description of the location. Return control to the main loop.

SEARCH

```
2820 REM ** search **
2830 IF P%<>24 THEN PRINT"I can't see anything!":R
ETURN
2840 IF SF=0 THEN SF=1:PRINT"I see something!":G$(
11)="a long rope with a hook attached.":N$(11)="ro
pe":RETURN ELSE PRINT"I see nothing!":RETURN
```

If you can remember back to the planning stage of the game, I decided to hide the rope under all the leaves in location 24. The rope and hook is to be object number 11, but the variables G$(11) and N$(11) were left blank when READing the DATA into the arrays, and these will need to be changed when the player searches through the leaves. Line 2830 informs the player that there is nothing there if he tries searching in any other location. The flag SF is set to one when the player has searched through the leaves so that he finds nothing if he tries to search through them again. The variables G$(11) and N$(11) are redefined if the rope is found.

Testing this routine merely involves moving to location 24 and searching the leaves. Try typing 'search' a second time to check that the computer tells you that there is nothing there.

Line

2830	if in any location other than number 24, print the message and return.
2840	if the flag is zero, change its value, print the message and change the description of object number 11, otherwise print the message and return to the main loop.

THROW

```
2850 REM ** throw **
2860 GOSUB 2240:IF R<>11 THEN PRINT"I don't see mu
ch point in that!":RETURN
2870 IF A(11)=0 THEN PRINT"I don't have it!":RETUR
N
2880 IF P%<>36 THEN PRINT"The hook doesn't catch o
n anything!":RETURN
2890 IF SG=0 THEN SG=1:PRINT"The rope catches on s
omething!":Q$(36)=Q$(36)+" A rope hangs down!"
2900 FOR X=1 TO 4:IF V$(X)=G$(11) THEN V$(X)=""
2910 NEXT:RETURN
```

The only method of moving into locations 37 and 38 is to throw the rope in room 36, so that it catches on the ring. The player will then be able to climb up the rope and enter the room of faces to get the jar of salt, which will be needed later in the game.

Line 2860 checks whether the player intends to throw anything else and returns control to the main loop if necessary. The value of A(11) is then checked to see whether the rope is being carried and finally the location is checked to make sure that there is only one place where the rope can hang.

The description of the location is changed, in line 2890, to include the information that the rope is hanging down, and an appropriate message is printed on the screen. Lines 2900 to 2910 are needed to remove the rope from the array V$(X), which contains the items being carried, so that the rope disappears from view! When you try to climb the rope later, the computer will check the value of the flag SG to see whether it has been set to 1 (line 2890).

Line

2860	call the subroutine which splits the sentence into two separate words and if the object mentioned is not number 11, print the message and return to the main loop.
2870	if the flag is zero, the player isn't carrying the rope, so the message is printed and control returned to the main loop.

2880	if the player is not in the correct location, print the message and return.
2890	if the flag is zero, change the value of the flag, print the message and change the description for the current location.
2900	search through the array V$(X), which holds the inventory of objects being carried, and remove the rope.
2910	return to the main control loop.

Setting the puzzles: part 3 6

CLIMB

```
2920 REM ** climb **
2930 GOSUB 2240
2940 IF P%=54 AND SL=1 THEN PRINT Y$:P%=49:RETURN
2950 IF R<>11 THEN PRINT"I can only climb a rope!"
:RETURN
2960 IF P%<36 OR P%>37 THEN PRINT"Not here!":RETUR
N
2970 IF SG<>1 THEN PRINT"Not just yet!":RETURN
2980 IF P%=36 THEN P%=37:PRINT Y$:RETURN
2990 P%=36:PRINT Y$:RETURN
```

The first line of this subroutine calls the subroutine to split the input sentence into two parts. Line 2940 then allows the player to climb the beanstalk in location 54 if, and only if, he has first planted it and then watered it, so setting the value of the flag SL=1.

The only other places where the player can climb up or down are locations 36 and 37. Line 2950 makes sure that the player actually types 'climb rope' and not just 'climb'. This line may be left out if you don't feel it necessary. Lines 2960 and 2970 then check the location and test to see if the rope is hanging from above (SG=1). Finally, the value of P% is changed in lines 2980 to 2990 depending on whether the player is climbing up or down the rope. Don't forget to check out the routine in the usual way before typing in the next one!

Line	
2930	call the subroutine to split the player's instructions into two words.
2940	if in location 54 and the beanstalk has grown, move to location 49, print the message and return to the main loop.
2950	if the object is not the rope, print the message and return to the main loop.

2960	if not in location 36 or 37, print the message and return to the main loop.
2970	if the rope has not caught on the ring, print the message and return.
2980	if in location 36, move to location 37, print the message and return.
2990	move to location 36, print the message and return.

GO UP

```
3000 REM ** go up **
3010 IF P%=70 THEN P%=71:PRINT Y$:RETURN
3020 IF P%=80 THEN P%=79:PRINT Y$:RETURN
3030 IF P%=36 AND SG=0 THEN PRINT"not just yet!":R
ETURN ELSE IF P%=36 THEN P%=37:PRINT Y$:RETURN
3040 IF P%=54 AND SL=1 THEN P%=49:PRINT y$:RETURN
3050 PRINT"I can't do that here!":RETURN
```

This subroutine is called whenever the player types 'go up' or simply 'up'. There are four locations where movement in this direction is possible and these are summarised below.

Location	*Description*	*Leads to location*
70	Inside barn	71 hayloft
80	Inside cavern	79 by grate
36	By ring	37 tunnel (if rope thrown)
54	Fertile land	49 beanstalk (if planted)

Line 3010 deals with the movement from location 70 to location 71, whilst line 3020 handles movement from location 80. Line 3030 checks the flag SG to see whether the rope is hanging down, before changing P% to 37 or printing a message to tell the player that he can't move in that direction. Similarly, line 3040 checks the value of SL to see whether the beanstalk has been planted and watered. Finally, if the program reaches line 3050, then the player is trying to 'go up' from a location where this is not possible! Do check out the routine in the usual way as soon as you have typed it in.

Line

3010	if in location 70, move to location 71, print the message and return.

3020 if in location 80, move to location 79, print the message and return.
3030 if in location 36 and the flag has not been set, print the message and return to the main control loop, otherwise move to location 37, print the message and return.
3040 if in location 54 and the flag has been set to indicate that the beanstalk has grown, move to location 49, print the message and return.
3050 print a message that it is not possible and return to the main loop.

DRINK

```
3060 REM ** drink **
3070 GOSUB 2240:IF R<>8 THEN PRINT"Don't be silly!
":RETURN
3080 IF A(8)=0 THEN PRINT"I don't have any!":RETUR
N
3090 E$="I drink the rum and in a drunken stupor,f
all and break my neck!":GOSUB 2610
```

This routine was written as a 'red herring'. The only object in the game which it is possible to drink is the bottle of rum, object 8. The first line of the routine checks whether the player has tried to drink anything else and prints a message about their stupidity if they have. Line 3080 then tests the value of A(8) to see if the rum is being carried and if, finally, the player is carrying it, then they trip in a drunken stupor and die! Ardent adventurers don't like games where they lose their lives too often, so don't go overboard with traps like this one and do try to keep the responses humorous. The main purpose of the rum in this game is to give to the tramp.

Line

3070 call the subroutine to split the sentence into two words and if the player is not referring to object number 8, the bottle of rum, print the message and return to the main loop.
3080 check to see if the player is carrying the rum and return to the main loop, after printing a suitable message, if not.
3090 set the message about getting drunk and call the lose game routine.

GIVE

```
3100 REM ** give **
3110 GOSUB 2240:IF R<>8 THEN PRINT"I'm not giving
";L$:RETURN
3120 IF A(8)=0 THEN PRINT"I don't have any!":RETURN
```

```
3130 IF P%<>69 THEN PRINT"There's nobody here who
would like it!":RETURN
3140 A(8)=0:FOR X=1 TO 4:IF V$(X)=G$(8) THEN V$(X)
=""
3150 NEXT:PRINT"The tramp thanks me and walks away
!"
3160 Q$(69)=LEFT$(Q$(69),40):SH=1:RETURN
```

This is a useful routine which allows you to give objects being carried to other people, or creatures, in the game. In this game, it is used only once — to give the bottle of rum to the tramp who will then walk away, so letting you go into the old barn. Line 3110 checks the number of the object you are trying to give and if this is not 8, the rum, it prints an appropriate message. The next check, in line 3120, is that you are carrying the rum and finally, line 3130 makes sure that you are in the correct location.

Once the computer has made sure that you want to give the rum, that you are carrying it and that you are in the right place, the bottle of rum is removed from the array V$(X), which holds the inventory and a suitable message is printed. The most important part of this routine is line 3160, where the flag SH is set to one, which is tested in the 'go in' routine previously described. Finally, the description of location 69 is shortened so that it no longer includes any mention of the tramp. This is because the tramp, like the wolf described already, is not a true object in this game. Testing this routine involves moving to location 25, getting the rum and taking it to location 69 to give to the tramp.

Line

3110	call the routine to split the player's instructions into two words and if the object being given away is not the rum, number 8, print the message and return to the main loop.
3120	check to see if the rum is being carried and print a suitable message before returning if not.
3130	if the player is not in location 69, print the message and return to the main loop.
3140	set the value of the flag to zero so that the computer knows that the rum is no longer being carried and then remove the description of the rum from the array V$(X) so that the inventory routine works correctly.
3150	print the message about the tramp.
3160	change the description for the current location, set the value of the flag SH to one and return to the main control loop.

USE

```
3170 REM ** use **
3180 GOSUB 2240
3190 IF R=4 AND A(4)=0 THEN PRINT"I don't have it!
":RETURN
3200 IF R=4 AND P%<>44 THEN PRINT"Nothing happens!
":RETURN
3210 IF R=4 THEN PRINT"The ghost disappears into t
he bag!":S%(44,2)=48:Q$(44)=LEFT$(Q$(44),29):RETUR
N
3220 IF R=14 AND A(14)=0 THEN PRINT"I don't have i
t!":RETURN
3230 IF R=14 AND P%<>33 THEN PRINT"There isn't muc
h point here!":RETURN
3240 IF R=14 THEN PRINT"The vampire flees for his
life leaving  something behind!":SI=1:G$(13)="a **
 JADE RING **":N$(13)="jade":RETURN
3250 IF R=16 AND A(16)=0 THEN PRINT"I don't have i
t!":RETURN
3260 IF R=16 AND P%<>35 THEN PRINT"There isn't muc
h point!":RETURN
3270 IF R=16 THEN PRINT"The slug shrivels up to no
thing and     leaves something on the ground.":G$(
15)="a ** SILK PURSE **":N$(15)="silk":SJ=1:RETURN

3280 IF R=22 AND A(22)=0 THEN PRINT"I don't have i
t!":RETURN
3290 IF R=22 AND (P%=35 OR P%=72 OR P%=62 OR P%=44
 OR P%=33) THEN E$="It explodes and covers me with
 a jet of flames!!":GOSUB 2610
3300 IF R=22 AND P%<>59 THEN PRINT"I don't see muc
h point in using it here!":RETURN
3310 IF R=22 THEN PRINT"The flames drive them away
!":S%(59,2)=61:Q$(59)=LEFT$(Q$(59),29):RETURN
3320 IF R=23 AND A(23)=0 THEN PRINT"I don't have i
t!":RETURN
3330 IF R=23 AND P%<>79 THEN PRINT"It's not much u
se here!":RETURN
3340 IF R=23 AND SM=0 THEN PRINT"The grate opens!"
:SM=1:Q$(79)=LEFT$(Q$(79),30)+" There is hole in t
he ground.":RETURN
3350 IF R=23 THEN PRINT"It's already open!":RETURN

3360 IF R=27 AND A(27)=0 THEN PRINT"I don't have i
t!":RETURN
3370 IF R=27 AND P%<>62 THEN PRINT"a sling is of l
ittle use here!":RETURN
3380 IF R=27 AND SN=1 THEN PRINT"I can't use it tw
ice!":RETURN
3390 IF R=27 AND SN=0 THEN SN=1:PRINT"That's done
the trick! The giant's body fades away. I see some
thing here!":G$(21)="an ** EMERALD **":N$(21)="eme
rald":RETURN
3400 PRINT"I can't use ";L$;" here!":RETURN
```

This is a very handy routine which allows the player to use many of the objects which he finds along the way. It is called from the main control section by typing either 'use' or 'prise'. In this game, there are six objects which can be used in this way. Line 3180 calls the routine which splits the input sentence into two words and stores the number of the object mentioned in the variable R. Careful study of the list will show that there are six different sections dealing with each of the objects separately. If R does not have the value 4,14,16,22,23 or 27, then the program will pass all the checks and reach line 3400 which tells the player that he can't use the object in question.

Object	*Used in Location*	*Purpose*
4 vacuum cleaner	44	To get rid of ghost
14 crucifix	33	To get rid of vampire
16 salt	35	To get rid of slug
22 flame thrower	59	To get rid of goblins
23 crowbar	79	To open the grate
27 sling	62	To get rid of giant

Use the vacuum cleaner

3190 check that it is being carried and return to the main loop if not.
3200 check the location to see if the ghost is there and return to the main loop if not.
3210 print the message, change the map for location 44 to allow movement south and change the description of the location.

Use the crucifix

3220 check that it is being carried and return to the main loop if not.
3230 check the location to see if the vampire is there and return to the main loop if not.
3240 print the message, change the value of the flag SI, change the vampire into the jade ring and return to the main loop.

Use the salt

3250 check that it is being carried and return to the main loop if not.
3260 check the location to see if the slug is there and return to the main loop if not.
3270 print the message, change the value of the flag SJ, change the descriptions of the slug into the silk purse and return to the main loop.

Use the flame thrower

3280 check that it is being carried and return to the main loop if not.
3290 call the lose game routine if in location 35, 72, 62 or 44.
3300 if not in location 59, return to the main loop.
3310 print the message, change the map to allow movement south from location 59, change the description of the location and return to the main loop.

Use the crowbar

3320 check that it is being carried and return to the main loop if not.
3330 return to the main loop if not in the right place (location 79).
3340 print the message about the grate opening, change the flag SM, which allows the player to go down, change the description of the location and return to the main loop.
3350 the grate must be open, so print an appropriate message and return to the main loop.

Use the sling
Note that the sling only appears in the game after the farmer has been given his dog back!

3360 check that the sling is being carried and return to the main loop if not.
3370 check the location to see if the giant is there and return to the main loop if not.
3380 check whether the sling has already been used.
3390 set the value of the flag SN, print the message, change the giant into the emerald and return to the main loop.
3400 this line is only reached if all the above tests have proved negative, so the player is told that he can't use the object and control is returned to the main loop.

The best method of testing this subroutine is to try out every possible combination catered for in the above lines. When you are entirely satisfied that everything works correctly, you could perhaps think about adding a few extra lines to this routine so that the player can 'use' other objects found within the game.

SWIM

```
3410 REM ** swim **
3420 IF P%=10 THEN P%=15:PRINT Y$:RETURN
3430 IF P%=15 THEN P%=10:PRINT Y$:RETURN
3440 IF P%=65 OR P%=68 THEN PRINT"The water's not
deep enough!":RETURN
3450 IF P%=64 OR P%=8 THEN E$="I drown....what a s
tupid suggestion!":GOSUB 2610
3460 PRINT"Don't be ridiculous!":RETURN
```

There are many occasions in adventure games where you want the player to move around by means other than walking. In this game, there are two locations where the player *must* swim to progress further. Swimming from location 10 takes the adventurer to location 15 and vice versa. This movement is taken care of in lines 3420 and 3430 respectively.

In locations 65 and 68, the water is not deep enough and the player will have to paddle across. Line 3440 deals with this. In two locations, numbers 64 and 8, the water is too dangerous to cross and the player drowns. This is dealt with in line 3450, where the lose game routine is called. There must be many other ways of losing your life when swimming (crocodiles, piranha etc.) and I'm sure that you could think up some terrible deaths for your victims. If the program reaches line 3460, then swimming is not possible in that location and a message to that effect is printed before control is returned to the main loop. By now, you should have got the hang of how to test that these routines are working properly.

Line	
3420	if in location 10, move to location 15 by changing the value of P%, print the message and return to the main loop.
3430	if in location 15, move to location 10 print the message and return to the main loop.
3440	if in location 65 or 68, print the message and return to the main loop.
3450	if in location 64 or 8, the contents of E$ are defined and the death routine is called.
3460	print the message about swimming being impossible and return to the main loop.

PADDLE

```
4140 REM ** paddle **
4150 IF P%=65 THEN P%=68:PRINT Y$:RETURN
4160 IF P%=68 THEN P%=65:PRINT Y$:RETURN
4170 IF P%=64 OR P%=10 OR P%=15 THEN PRINT"The wat
er's too deep!":RETURN
4180 PRINT"I can't go paddling here dummy!":RETURN
```

In this game, we have already decided to make it impossible to swim from location 65 to 68 and vice versa because the water is too shallow. Thus lines 4150 and 4160 allow the player to paddle across the river between these two locations. In locations 10, 15 and 64, the water is too deep and the player cannot paddle (line 4170). You may like to change this line so that the player gets attacked by strange fish and loses the game. Line 4180 is only reached if the player tries to paddle in any other location and therefore a message to that effect is printed!

Line

4150	if in location 65, move to location 68, print the message and return to the main loop.
4160	if in location 68, move to location 65, print the message and return to the main loop.
4170	if in location where the water is too deep, print the message and return.
4180	print a message about paddling being impossible and return to the main loop.

UNLOCK

```
3470 REM ** unlock **
3480 IF A(3)=0 THEN PRINT"I have no key!":RETURN
3490 IF SK=0 AND P%=16 THEN PRINT"The lock's too r
usty!":RETURN
3500 IF P%=16 THEN PRINT Y$:PRINT"The chain comes
loose.":Q$(16)=LEFT$(Q$(16),40):S%(16,2)=17:RETURN
3510 IF P%=79 THEN PRINT"There's no keyhole!":RETU
RN
3520 PRINT"Don't be silly!":RETURN
```

There is only one lock in this game which can be unlocked and you'll need to have oiled it first. The key, object number 3, is found in location 1 and unless A(3)=1 then the player is not carrying it (line 3480). The lock is to be found in location 16 and a check is made on the value of the flag SK in line 3490 to see whether the lock has been oiled first. Unless SK has been set, it is impossible to unlock the gate and line 3500 is not reached. Line 3500 prints a message about the chain, alters the description of location 16 and changes the map so that the player can move south from that location. A metal grate is set into the ground in location 79 and the player may try to unlock this with the same key. This is checked for in line 3510. There are no other locations within this game where the player could reasonably try to unlock anything and therefore the message 'Don't be silly!' is printed for all other attempts to unlock anything (line 3520). It isn't possible to check that this routine works properly until you have typed in the subroutine to 'oil' the lock.

Line

3480	check the value of the flag to make sure that the player is carrying the key and return to the main loop if not.
3490	if in location 16 and the flag has not been set, the gate has not been oiled and so the message is printed and control returned to the main loop.
3500	if in location 16, print the message, change the description of the location and return to the main loop.
3510	if in location 79, print the message and return.
3520	print the message about it being impossible and return to the main loop.

OIL

```
3530 REM ** oil **
3540 IF P%<>16 THEN PRINT"I can't!":RETURN
3550 IF A(2)=0 THEN PRINT"no oil!":RETURN
3560 PRINT Y$:SK=1:RETURN
```

Before the player can unlock the gate found in location 16, the lock must be oiled. The first check, in line 3540, is that the player is in the correct location. A check is then made in line 3550 on the value of A(2) to make sure that he is carrying the oil. Finally, the flag SK is set to one when the lock has been oiled (line 3560). The value of SK is checked when trying to unlock the gate and hence you will need to have typed in both routines (oil and unlock) before you are able to test whether they work correctly. Testing this section involves going into the Wizard's cottage and getting the oil. En route to location 16, the player must get the key, and on reaching the gate, try to unlock it. Once unlocked, going south from location 16 should take the player to location 17. The easiest way of testing this is to try going south and then escape from the program. You can then type PRINT P% and the value 17 should be printed on the screen.

Line

3540	if not in location 16, print the message and return to the main loop.

3550 check to make sure that the player is carrying the oil and return to the main loop if not.
3560 print the message, set the value of the flag and return to the main loop.

PLANT

```
3570 REM ** plant **
3580 IF A(1)=0 THEN PRINT"I can't!":RETURN
3590 IF P%<>54 THEN PRINT"The ground's too hard!":
RETURN
3600 FOR X=1 TO 4:IF V$(X)=G$(1) THEN V$(X)="":PRI
NT Y$
3610 NEXT:G$(1)="a tiny little beanstalk murmuring
.......water,water!":B%(1)=54:N$(1)="":A(1)=2:RETU
RN
```

One of my favourite puzzles in the original adventure involved watering the beanstalk and climbing up the branches into a new area of caverns. I have taken this idea to show you how you can incorporate such puzzles within your own game.

Line

3580 check the value of A(1) to see if the player is carrying the beanstalk.
3590 check the location. The only place where the beanstalk can grow is in location 54 and so control is returned to main loop if the player is in the wrong place.
3600 check through all the items being carried (V$(X)) and remove the beanstalk.
3610 change the description of the beanstalk, change the value of B%(1) to drop the beanstalk in location 54, change the value of N$(1) so that the computer no longer recognises the word 'beanstalk', set the flag A(1)=2 so that the computer knows that the beanstalk has been planted and return to the main loop.

It is very important that N$(1) is emptied so that the player is unable to pick up the beanstalk after it has been planted. This routine cannot be tested until the routine to 'pour water' has been typed into your computer.

FILL

```
3620 REM ** fill **
3630 IF A(5)=0 THEN PRINT"Fill what ?":RETURN
3640 IF P%=10 OR P%=15 OR P%=26 OR P%=64 OR P%=65
OR P%=68 THEN PRINT Y$:A(5)=2:RETURN ELSE PRINT"I
can't do that here!":RETURN
```

When the player has planted the beanstalk, it will start murmuring 'water, water' and the next task is to find the vase and fill it with water. The vase, object number 5, is found in location 25 and can be filled at the kitchen sink, location 26, or on the river banks, locations 64, 10, 15 or 68.

Line

3630	check that the vase is being carried and return if not.
3640	check the location and change the value of A(5) to 2 if the vase can be filled.

A(5) will usually have the value 1 if it is being carried and therefore once it has been filled with water, its value is set to 2. If the player drops it once it has been filled, the value of A(5) will go back to zero and the water will spill!

POUR

```
3650 REM ** pour **
3660 IF A(5)<>2 THEN PRINT"I can't!":RETURN
3670 PRINT Y$:A(5)=1
3680 IF P%<>54 OR A(1)<>2 THEN RETURN
3690 IF SL=0 THEN PRINT"The beanstalk spurts into
rapid growth!":G$(1)="an enormous beanstalk reachi
ng high     into the clouds.":SL=1
3700 RETURN
```

The final part of this puzzle involves taking the vase, which should be full of water, to location 54 and pouring it onto the beanstalk.

Line

3660	check whether the player is carrying a vase full of water and return if not.
3670	print the message 'O.K.' and empty the vase by setting A(5) to 1.
3680	if the location is not number 54, or the beanstalk has not been planted then return to the main control loop.
3690	check the value of SL to make sure that the beanstalk has not already grown, print the message, change the description of the beanstalk, set the value of the flag SL.
3700	return to the main program control loop.

The value of the flag SL is tested in the routine to climb the beanstalk and therefore when you have checked that these three routines (plant, fill and pour) are working, you would be advised to escape from the program and type PRINT SL. This should return the number 1 if you are to be able to climb the beanstalk later.

Setting the puzzles: part 4 7

GO DOWN

```
3710 REM ** go down **
3720 IF P%=37 THEN P%=36:PRINT Y$:RETURN
3730 IF P%=49 THEN P%=54:PRINT Y$:RETURN
3740 IF P%=71 THEN P%=70:PRINT Y$:RETURN
3750 IF P%=79 AND SM=0 THEN PRINT"I can't get past
 the grate!":RETURN
3760 IF P%=79 THEN P%=80:PRINT Y$:RETURN
3770 PRINT"I can't!":RETURN
```

This routine complements the routine to 'go up' and the two routines should be tested together.

Line

3720	if the player is in location 37, change location to 36 and return to the main loop.
3730	if the current location is 49, change it to 54 and return to main loop.
3740	if the current location is 71, move to location 70 and return to the main loop.
3750	if the location is 79 and the grate is closed (SM=0), then print the message and return to the main loop.
3760	if the location is 79, the grate must be open, so move to location 80 and return to the main loop.
3770	if this line is reached, the player is in a location where he can't go down, so the message is printed and control returned to the main loop.

PRESS

```
3780 REM ** press **
3790 IF P%<>64 THEN PRINT"I can't do that here!":R
ETURN
3800 PRINT"There are three buttons."
```

```
3810 PRINT"RED  GREEN  and BLUE"
3820 INPUT "Which one do I press ";Z$:Z$=LOWER$(Z$
)
3830 Z$=LEFT$(Z$,1):IF SC=1 THEN E$="A snake crawl
s from behind the buttons  and sinks its fangs int
o me!":GOSUB 2610
3840 SC=1:IF Z$<>"b" THEN PRINT"Nothing seems to h
appen!":SC=0:RETURN
3850 PRINT"A door opens!":RETURN
```

On reaching location 64, the player will come across a row of three buttons. Pressing the correct one will open the door into the statue, but the wrong one can be dangerous! This is the only occasion in the whole game where the player needs to press anything.

Line

3790	check the location and return to the main loop if it isn't 64.
3800–3810	describe the colours of the three buttons to the player.
3820	input the player's choice.
3830	if the correct button has already been pressed, the snake will attack, and the lose game routine is called.
3840	if the player doesn't press the blue button, nothing happens and control is returned to the main loop.
3850	open the door and return to the main loop.

You will notice that the flag SC is set to 1 if the door opens, and you may like to change this section so that a random colour must be chosen, or perhaps giving the player just two attempts to get it right. This section should be tested by moving to location 64 and pressing the buttons. Do make sure that pressing the blue button twice does kill the player.

We have now finished the main part of the program and it should be possible for you to solve the game as it stands. There is still plenty of memory left for you to add extra puzzles and problems of your own, although the final two subroutines will take up much of this spare RAM. A routine which allows you to save your position onto tape and load it back in again later is an extremely useful feature of any adventure and can transform your program into a very professional piece of programming.

SAVE GAME

```
3860 REM ** save game **
3870 PRINT"Please insert tape/disc now "
3880 OPENOUT"data"
3890 FOR X=1 TO 80:PRINT#9,Q$(X):NEXT X
```

```
3900 FOR X=1 TO 80:FOR Y=1 TO 4:PRINT#9,S%(X,Y):NE
XT Y,X
3910 FOR X=1 TO 30:PRINT#9,G$(X):NEXT X
3920 FOR X=1 TO 30:PRINT#9,B%(X):NEXT X
3930 FOR X=1 TO 30:PRINT#9,N$(X):NEXT X
3940 FOR X=1 TO 30:PRINT#9,N%(X):NEXT X
3950 FOR X=0 TO 30:PRINT#9,A(X):NEXT X
3960 FOR X=1 TO 4:PRINT#9,V$(X):NEXT X
3970 PRINT#9,SA,SB,SC,SD,SE,SF,SG,SH,SI,SJ,SK,SL,S
M,SN,SO,SP,P%
3980 CLOSEOUT:RETURN
3990 REM
```

When writing any save game routine, we must open a cassette or disc file and save to it the values of any variables whose value might have changed during the course of the game. In this program, there are a few locations where the descriptions remain constant, but most variables can change their value. For this reason, I decided that the easiest way of writing the routine was to save the value of all variables used onto the tape (or disc) using the PRINT #1 command.

Line

3870	print the message asking the player to insert the tape into the recorder.
3880	open the file with the filename 'data'.
3890	write the descriptions of the 80 locations onto the tape.
3900	write the details of the map onto the tape.
3910	write the descriptions of the objects onto tape.
3920	write the locations where the objects can be found onto tape.
3930	write the words recognised by the computer onto tape.
3940	write the pointers to the words onto tape.
3950	write the flags for the objects being carried onto tape.
3960	write the descriptions of the objects being carried onto tape.
3970	write the flags SA - SP and the current position (P%) onto tape.
3980	close the file and return to the main loop of the program.

It is extremely important when writing this routine that you don't forget to include any of the variables you have introduced as flags. It's all too easy to forget to include one and you would be well advised to write them all down on paper as you introduce them into the game.

LOAD GAME

```
4000 REM ** load game **
4010 PRINT"Please insert tape/disc now "
4020 OPENIN"data"
4030 FOR X=1 TO 80:INPUT#9,Q$(X):NEXT X
4040 FOR X=1 TO 80:FOR Y=1 TO 4:INPUT#9,S%(X,Y):NE
XT Y,X
4050 FOR X=1 TO 30:INPUT#9,G$(X):NEXT x
4060 FOR X=1 TO 30:INPUT#9,B%(X):NEXT X
4070 FOR X=1 TO 30:INPUT#9,N$(X):NEXT X
4080 FOR X=1 TO 30:INPUT#9,N%(X):NEXT X
4090 FOR X=0 TO 30:INPUT#9,A(X):NEXT X
4100 FOR X=1 TO 4:INPUT#9,V$(X):NEXT X
4110 INPUT#9,SA,SB,SC,SD,SE,SF,SG,SH,SI,SJ,SK,SL,S
M,SN,SO,SP,P%
4120 CLOSEIN:RETURN
4130 REM
```

Having saved a game onto tape, the next routine required will be the one to load it back into memory. The purpose of this subroutine is to restore the value of all of the variables back to that when the game was saved. In many ways, this routine is a mirror image of the save game routine. After opening the file, the variables must be read back in from tape in exactly the same order as they were saved. Any errors, however slight, in the ordering of the variables will cause disaster.

Line	
4010	print the message asking the player to insert the tape.
4020	open channel to input the cassette file.
4030	input the descriptions of the 80 locations.
4040	input the data for the map.
4050	input the descriptions of the 30 objects.
4060	input the positions where the objects are to be found.
4070	input the words understood.
4080	input the pointers to the words recognised.
4090	input the flags to tell the computer which items are being carried.
4100	input the descriptions of the items being carried.
4110	input the flags SA-SP and the current location (P%).
4120	close the file and return to the main program.

Once you have typed in the two routines to save a game and load it back in again, you should check that it does in fact work. If any errors occur when trying to load the tape back into the computer, there are a number of possible causes. I have summarised these below.

1 Trying to load back the tape when the OPENIN command uses a different filename. Check that both routines open the file with the name 'data'.
2 The tape is faulty. Try saving a new version of your game on a different tape.
3 The order in which the data is read in from tape is different from the order in which it was saved. Check the listing of the two routines very carefully.
4 You have saved a description of a location which contains a comma. This may cause the computer to think that it is loading in the next item of data. Make sure that you don't allow the contents of any array to contain a comma. If you must have a description in your game containing a comma, then you should change the comma to another symbol such as a percentage sign and then change it back to a comma again after the tape or disc has been loaded or saved.

Using this technique to store the data for an adventure game is very inefficient in memory usage. The data is stored twice, once in the data lines and again on the data tape. If you really want to make maximum use of the memory available, then you would be advised to load the data in from tape or disc every time. This would, of course, mean that you would have to write a separate program to create the first data tape of all. If you do attempt to write a program in this way, you will also find that every time you make a simple mistake, then you will have to load the data file in again and this in turn means that the development time will be much greater. Later on in the book, I shall introduce a game which was written in this way and in which the program to create the first data file allows you to change the program by answering a few questions!

Now that we have completed *The Wizard's Quest*, you might like to try extending the game by including a few extra puzzles and problems for the player to pit his wits against. There is, after all, still plenty of RAM left in the Amstrad to make use of some of the objects found in the game which I haven't made much use of. You should, by now, be feeling fairly confident about how the program works and might like to try adding a few extra subroutines. Here are a few suggestions which you might like to try out.

1 The flame thrower refuses to work until you find some fuel. This could be done by waving the wand over the top of the pile of leaves.
2 The vacuum cleaner is broken and you need to repair it before it will work. You might need to use the rubber gloves as a new rubber belt (after you have got into the chapel of course).
3 The lid to the jar of salt is stuck and you need to get help to open it. Perhaps the farmer might oblige.

I'm sure you can think of plenty of other puzzles, especially if you can introduce more objects of your own into the game. Do remember that you will need to change the numbers in the get, drop, check, load and save routines if you do add extra objects.

8 Making life difficult

Writing adventures for yourself is great fun, but you'll get a far greater sense of achievement if you can write an adventure which can be shared with others. One of the unfortunate features of BASIC as a language for adventure games is that you can escape from the program and list it. It is always far easier to solve a game by examining the listing than actually playing the game, and whenever the player comes across a tricky problem, there is always the temptation to cheat. What can you, as the writer of the game, do about it? How can you make life harder for the player?

There are several approaches which can be adopted and your choice of technique will depend on whether you are writing a game for your friends in the local computer club, for sale to a software house or for publication in a magazine. One of the most useful techniques is to code all the data lines so as to make the program more difficult to solve. There are many different ways of doing this and the listing below illustrates just one of them. Try typing it in and running it.

```
10 DATA "In a dark and gloomy forest."
20 READ A$
30 FOR x=1 TO LEN(A$)
40 B$=B$+CHR$(ASC(MID$(A$,X,1))+1)
50 NEXT X
60 PRINT#8,"Original string :-"
70 PRINT#8,A$
80 PRINT#8:PRINT#8
90 PRINT#8,"Final string :-"
100 PRINT#8, B$

Sample Run

Original string :-
In a dark and gloomy forest.


Final string :-
Jo!b!ebsl!boe!hmppnz!gpsftu/
```

This short program illustrates how we can change a line of DATA to make it much more difficult for the player to decipher. What we have done is to shift all the letters along the ASCII code by one.This is done by looking at each of the letters of A$ in turn, finding the ASCII code of it, adding one to the ASCII code and then printing the appropriate character string using CHR$. Thus letter 'a' becomes 'b', 'b' becomes 'c' etc. The ASCII code for a space is 32, so that adding one to it and finding the corresponding character, produces '!'. There is no reason why you should be limited to shifting the characters along the ASCII scale by just one. If you want to try shifting them by 3, then you should change line 40 to

40 B$=B$+CHR$ASC((MID$(A$,X,1))+3)

Using this technique in practice requires a little care. The first thing you will have to do is to change all the descriptions in the DATA lines to their coded format. You will need to be exceptionally careful of errors at this stage because it's extremely difficult to spot spelling mistakes when the data has been coded. I would suggest that you let the computer work out the coding for you, using the routine printed above, rather than try to code it manually. You can then change the data in your program by using the editing facilities of your micro. This is a very laborious process and you may well think that the effort is not worth while. Once you have coded all the data lines containing the descriptions of the locations, you will need to insert a few extra lines into your program to decode them. The coding to do this is, in principle, exactly the opposite of the listing we used to produce the coding and therefore you may like to try working out how it works.

```
1101 H$="":I$=Q$(P%)
1102 FOR X=1 TO LEN(I$)
1103 H$=H$+CHR$(ASC(MID$(I$,X,1))-1)
1104 NEXT X
1105 Q$(P%)=H$
```

If you have gone this far to make life more difficult for the cheat, then you will probably want to use a similar technique to code the descriptions of the objects, the words understood and the messages. There is no reason why you shouldn't use a different code for each section of data to make it even more difficult for the player to crack. You can use the same coding and decoding lines as before, although you will have to make changes to the names of the strings being decoded.

If you intend to sell your program to other enthusiasts, or to a software house, then coding your program in this way is well worth the effort. Programs listed in magazines, on the other hand, are often typed in by complete beginners and editors of all computer magazines will be able to tell you of the many letters received complaining about programs which don't work. In practice, most magazines print listings directly from working copies of programs and therefore the vast majority of these errors are caused by typing mistakes on the part of the readers. Most editors prefer programs which are not going to cause too many problems for their readers!

If your program is well structured, it should be very easy to solve the adventure by merely examing the listing and although a game written using spaghetti programming would be much more difficult to solve from the listing, you will find it far more satisfying to write a structured program and code the data lines to prevent cheating.

Many commercial adventure programs are written in specially created adventure languages. The most famous of these is called A–CODE and was written by Mike Austin to help in the production of the excellent series of adventures from Level Nine Computing. This code combines all the advantages of machine code speed with data compression techniques to produce adventures which are truly amazing. In *Snowball,* for example, they have managed to cram over 7000 locations, 700 different messages and 60 objects into only 32K of memory. This has been achieved by using a coding system which replaces many common words such as 'the' with single characters. The result of this is that text messages can be compress-ed into less than half of their original size, which means that the games can be far more detailed. Such techniques are beyond the scope of this book, but should provide avenues for exploration for the more advanced programmer. Using data compression does also have the advantage of making a game almost impossible to solve by examining the listing.

A method of protecting your program which is a little easier to implement is to add protection to the program when saving it to tape. This can be done by typing SAVE"ADVENTURE",P. A program protected in this way can be sold on tape through the usual channels, but please don't try submitting such a program to a magazine editor! It's important to remember to keep an unprotected copy of your program as well, just in case you want to edit the program at a later date.

Snow White: part 1 9

A graphic adventure is a very different type of program from a text only adventure and must be planned in a totally different way. Snow White is an adventure game for young children and contains a full high resolution picture of each location visited. A number of sound effects have also been incorporated within the game and these too were decided upon before programming started rather than added as an afterthought.

The starting point for this game, as with most adventures, is the map. In a game designed for younger children, it is important to keep the sentences used for descriptions fairly short and to make the pictures as bright and colourful as possible. Despite the advanced facilities available for the Amstrad CPC464 and 664 to help with graphic design, the pictures used in this game do take up a vast amount of user memory. In addition, the FILL command is not available on the CPC464, although it is on its larger brother, the CPC664. In order to make the game compatible with both, I have not used the FILL command (well it was written on a CPC464 !). For these reasons, I decided to include only 24 locations within the game. In many graphic adventures, including *The Hobbit,* the programmers have chosen to include graphics for only a few locations, which allows them to pack more into the game. Even in *The Hobbit,* however, there are only just over double the number of locations found in this game.You must choose which path to follow right from the start. My own feeling is that a game designed for younger children should include as many pictures as possible, whilst programs aimed at older enthusiasts should place greater emphasis on text.

Once you have designed the map for your game and have put a brief description of each location alongside the corresponding box, you have a choice of directions to follow. You can either convert the map into data lines, as we did with the last program, and develop the graphics later, or you can go straight to the graphics. In most cases it is easier to develop the pictures first because you will then have a better idea of the amount of memory left for the rest of the game. In practice, however, there is little difference between the two approaches.

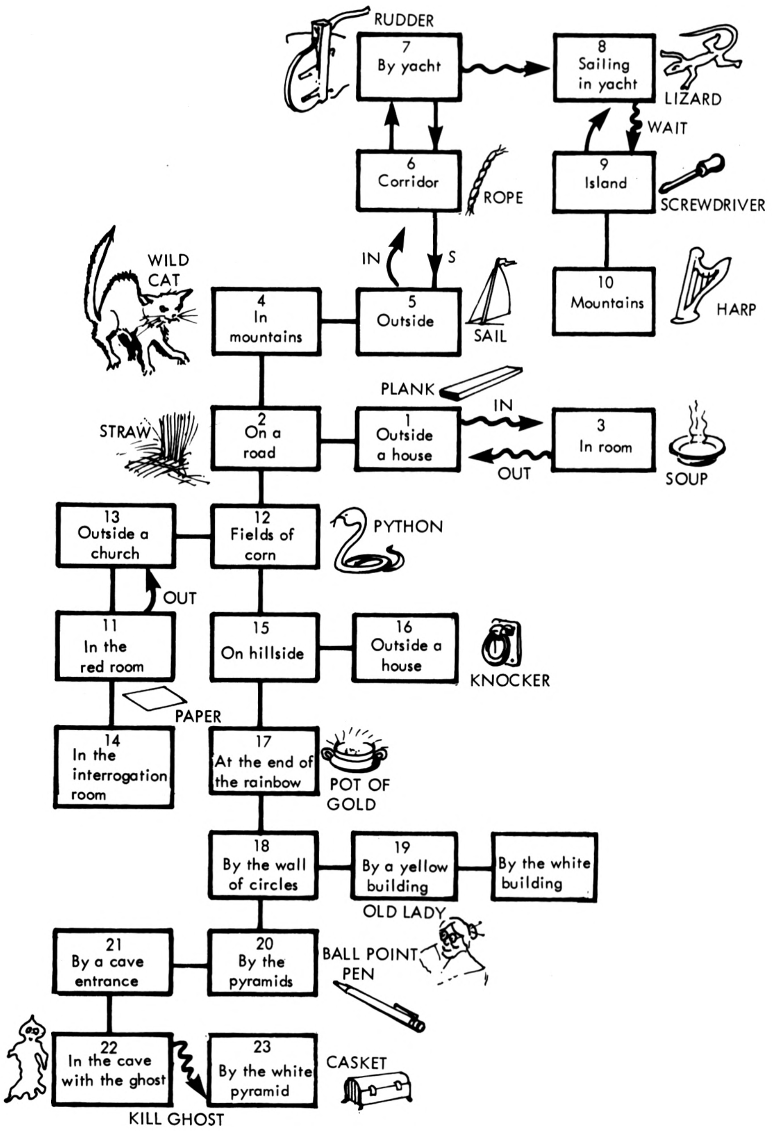


Fig. 9.1 The map for *Snow White.*

In this game, all the pictures are drawn in MODE 0 and the text is written in MODE 1. The reason for doing things this way is that it allows more colours to be displayed at once, whilst still allowing the text to be very legible. We must take care when changing from MODE 0 back to MODE 1 to set the paper and pen colours back to their default colours, otherwise we may well end up with yellow text on a yellow background.

In location 22, where there is some animation in the graphics, I have used a large graphics character built up out of four user defined characters. The definition of ghost$ is done right at the start of the program and will be discussed later.

When I introduce the flowchart, you will notice that the program is constantly changing between the two modes and, in order to make life easier, I have directed the entry to the graphics section through a subroutine which controls the graphics display.

Subroutine to control the graphics

```
2250 MODE 0
2260 ON P% GOSUB 2350,2440,2530,2590,2640,2680,273
0,2790,2840,2900,2960,3040,3120,3180,3260,3340,343
0,3490,3550,3630,3680,3720,3750,
3800
2270 MODE 1
2280 RETURN
2290 END
```

Line

2250	selects MODE 0 to allow more colours to be displayed.
2260	examines the value of P%, the current location and calls the correct subroutine.
2270	return to MODE 1 for the text.
2280	return to the main program control loop.

Whenever this subroutine is called, the program examines the value stored in P% and calls the subroutine for that particular location. Thus if the player is in location 5, the value of P% will also be 5 and the program will call the fifth subroutine in line 2260. In this way, the graphics instructions held from line 2640 onwards will display the picture for location 5. Each time you type in the graphics instructions for a new location, you should check it to make sure that it works properly. If, for example, you have just completed the graphics for location 7, you should adopt the following procedure.

1 Type P%=7 and press ‹ENTER›
2 Type GOSUB 2250 and press ‹ENTER›

Line 2260 will then call the subroutine at line 2730 and the computer should display the picture associated with location 7. Before you reach this stage, however, it is very important that you type in the short listing below. This is used to prevent the program returning from display of the picture to the text section until the player has pressed the space bar.

Preventing return to MODE 1

```
2300 WINDOW #2,2,19,22,24:PAPER #2,5: PEN #2,4:CLS
 #2:LOCATE #2,1,2:PRINT #2,"Press  <Space Bar>"
2310 F$="":WHILE F$<>" ":F$=INKEY$:WEND
2320 RETURN
```

Line

2300	defines a text window, sets its paper colour to black and prints the message in it.
2310	waits for the space bar to be pressed.
2320	return to graphics control section.

Notice that I have used a WHILE WEND loop in line 2310 to replace the more usual

2310 A$=INKEY$:IF A$< >" " THEN 2310

Now that you have typed the two sections of code needed to control the graphics, you can press ahead with the graphics themselves. I have split the pictures up into sections to make it easier to follow. If you type in each subroutine separately and test it out, rather than leaving debugging until the end, it should make it easier to find and rectify any errors.

Rather than spend a long time here describing how the graphics commands work, I will leave you to type them in and try them out for yourself. If you would prefer to design your own graphics, then you should refer to Chapter 18 where the subject is dealt with in greater depth.

Location 1. Outside the small house

```
2330 REM ** graphics for locations **
2340 REM ** location 1 **
2350 PAPER 0:CLS:FOR Y=110 TO 202:MOVE 30,Y:DRAWR
300,0,1:NEXT Y
2360 FOR Y=202 TO 240:MOVE 180,Y:DRAWR 200-(Y-100)
*0.4,0,3:MOVE 180,Y:DRAWR (Y-100)*0.4-200,0,3

2370 NEXT Y:FOR Y=230 TO 270:MOVE 240,Y:DRAWR 10,0
,2:NEXT y
2380 FOR Y=112 TO 130:MOVE 290,Y:DRAWR 30,0,3:MOVE
 290,Y+50:DRAWR 30,0,3
2390 MOVE 120,Y:DRAWR 30,0,3:MOVE 60,Y:DRAWR 30,0,
3:NEXT
2400 X%=500:Y%=300:COL%=4:M=3:GOSUB 3890
2410 FOR Y=110 TO 120:MOVE 331,Y:DRAWR 289,0,12:NE
XT
2420 GOSUB 2300:RETURN
```

Location 2. On the wide road

```
2430 REM ** location 2 **
2440 PAPER 2: CLS:WINDOW #1,1,20,10,25:PAPER #1,12
:CLS #1
2450 FOR y=200 TO 300:MOVE 300,y:DRAWR 200,0,3:NEX
T
2460 Z=0:FOR Y=301 TO 330: MOVE 400,Y:DRAWR 120-Z,
0,5:MOVE 400,Y: DRAWR Z-120,0,5:Z=Z+4:NEXT Y
2470 FOR Y=200 TO 240:MOVE 400,Y:DRAWR 20,0,1:MOVE
 450,Y:DRAWR 20,0,1:MOVE 470,Y+100:DRAWR 15,0,7:NE
XT Y
2480 FOR Y=250 TO 265: MOVE 310,Y:DRAWR 50,0,2:MOV
E 440,Y:DRAWR 50,0,2:NEXT Y
2490 FOR Y=0 TO 185:MOVE 395+Y/10,Y:DRAWR 101-Y/2,
13:NEXT
2500 X%=200:Y%=300:COL%=4:M=3:GOSUB 3890
2510 GOSUB 2300:PAPER 0:RETURN
```

Location 3. In the small room

```
2520 REM ** location 3 **
2530 PAPER 0:CLS:WINDOW #1,4,17,7,19:PAPER #1,13
2540 FOR Y=1 TO 200:MOVE 0,Y:DRAWR Y,Y/100,7:MOVE
0,400-Y:DRAWR Y,Y/100,7:NEXT Y
2550 FOR Y=1 TO 200:MOVE 640,Y:DRAWR -Y,Y/100,7:MO
VE 640,400-Y:DRAWR -Y,Y/100,7:NEXT Y
2560 CLS #1:X%=320:Y%=200:COL%=5:GOSUB 3890
2570 GOSUB 2300:PAPER 0:RETURN
```

Location 4. In the misty mountains

```
2580 REM ** location 4 **
2590 PAPER 10:CLS:FOR Y=1 TO 120:MOVE 0,y:DRAWR 64
0,0,5:NEXT:FOR Y=1 TO 120:MOVE 0,Y:DRAWR 220+Y/3,0
,9:MOVE 640,Y:DRAWR -220-Y/3,0,9
:NEXT
2600 FOR Y=121 TO 400: MOVE 50+Y/3,Y:DRAWR 400-Y,0
,12:NEXT:INK 13,9:FOR Y=121 TO 320:MOVE 300+Y/2,Y:
DRAWR 500-y,0,13:NEXT
2610 FOR Y=121 TO 370:MOVE 0,Y:DRAWR 400-Y,0,13:NE
XT Y:X%=550:Y%=380:COL%=1:GOSUB 3890
2620 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 5. Outside the cavern of light

```
2630 REM ** location 5 **
2640 PAPER 5:CLS:INK 13,9:FOR A=1 TO 180:DEG:PLOT
320,200
2650 DRAW 320+190*COS(A),200+190*SIN(A),4:NEXT:FOR
```

```
 Y=0 TO 210:MOVE 0,Y:DRAWR 640,0,13:NEXT Y
2660 FOR Y=1 TO 210:MOVE 190+Y/1.5,Y:DRAWR Y-200,0
,1:NEXT:GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 6. In a narrow corridor

```
2670 REM ** location 6 **
2680 PAPER 9:CLS:WINDOW #1,9,11,12,14:PAPER #1,4:C
LS #1
2690 MOVE 0,0:DRAW 260,180,4:MOVE 640,0:DRAW 345,1
80,4
2700 MOVE 0,400:DRAW 260,220,4:MOVE 640,400:DRAW 3
45,220,4
2710 GOSUB 2300:PAPER 0:RETURN
```

Location 7. On the sea shore

```
2720 REM ** location 7 **
2730 PAPER 10:CLS:INK 13,19:FOR Y=1 TO 120:MOVE 0,
Y:DRAWR 640,0,13:NEXT: FOR Y=121 TO 171:MOVE 100,Y
:DRAWR 200,0,0:MOVE 100,Y:DRAWR
-Y/2,0,0:MOVE 300,Y:DRAWR Y/2,0,0:NEXT Y
2740 FOR Y=0 TO 165:MOVE 550,Y:DRAWR 100,0,5:NEXT
Y:FOR Y=1 TO 2:MOVE 550,165+Y:DRAWR -200,Y*2,7:NEX
T Y
2750 FOR Y=1 TO 4:MOVE 200+Y,171:DRAWR 0,150,7:NEX
T Y:FOR Y=191 TO 291:MOVE 205,Y:DRAWR 291-Y,0,3:NE
XT: FOR Y=192 TO 305: MOVE 199,Y
:DRAWR Y-305,0,3:NEXT Y
2760 X%=500:Y%=350:COL%=1:GOSUB 3890
2770 GOSUB 2300:INK 13,22:PAPER 0:RETURN
```

Location 8. In the yacht

```
2780 REM ** location 8 **
2790 PAPER 0:CLS:FOR R=1 TO 360:DEG:MOVE 320,200:D
RAWR 100*COS(R),100*SIN(R),4:NEXT
2800 FOR Y=1 TO 150:MOVE 0,Y:DRAWR 640,0,6:NEXT Y
2810 INK 13,9:WINDOW #1,10,11,12,12:PAPER#1,13:CLS
#1
2820 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 9. On a small island

```
2830 REM ** location 9 **
2840 PAPER 2:CLS:FOR R=1 TO 90:DEG:MOVE 320,200:PL
OT 320+190*COS(R),200+190*SIN(R),5:DRAW 640,200+19
0*SIN(R),5:NEXT R
2850 FOR R=90 TO 180:DEG:MOVE 320,200:PLOT 320+190
```

```
*COS(R),200+190*SIN(R),5:DRAW 0,200+190*SIN(R),5:N
EXT R
2860 INK 13,9:FOR Y=1 TO 200:MOVE 0,Y:DRAWR 640,0,
13:NEXT:FOR Y=380 TO 400:MOVE 0,Y:DRAWR 640,0,5:NE
XT
2870 FOR Y=1 TO 200:MOVE 550-Y,Y:DRAWR -400+Y*2,0,
9:NEXT
2880 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 10. On a mountain pass

```
2890 REM ** location 10 **
2900 PAPER 5:CLS:INK 13,9:INK 11,22
2910 FOR Y=300 TO 400:MOVE 0,Y:DRAWR 640,0,2:NEXT
Y:FOR Y=100 TO 360:MOVE 100+Y/3,Y:DRAWR 350-Y,0,13
:NEXT
2920 FOR Y=400 TO 90 STEP -1:MOVE 0,Y:DRAWR 420-Y,
0,12:NEXT Y
2930 FOR Y=390 TO 120 STEP -1:MOVE 280+Y/2,Y:DRAWR
 -y+390,0,13:NEXT Y:FOR Y=380 TO 85 STEP -1:MOVE 6
40,Y:DRAWR Y-400,0,12:NEXT Y
2940 GOSUB 2300:PAPER 0:INK 13,22:INK 11,16:RETURN
```

Location 11. In a strange room

```
2950 REM ** location 11 **
2960 INK 13,3:PAPER 13:CLS:WINDOW #1,8,12,11,15:PA
PER #1,3:CLS #1
2970 MOVE 0,0:DRAW 260,180,3:MOVE 640,0:DRAW 345,1
80,3
2980 MOVE 0,400:DRAW 250,220,3:MOVE 640,400:DRAW 3
50,220,3
2990 FOR Y=400 TO 240 STEP -1:MOVE 320,Y
3000 DRAWR (Y/3-80)*4.8+50,0,5
3010 MOVE 320,Y:DRAWR -(Y/3-80)*4.2-90,0,5:NEXT Y
3020 GOSUB 2300:PAPER 0:RETURN
```

Location 12. In a field of corn

```
3030 REM ** location 12
3040 INK 13,9:PAPER 13:CLS:DEG:FOR R=180 TO 360 ST
EP 0.4:MOVE 320,400
3050 DRAWR 400*COS(R),300*SIN(R),2:NEXT R
3060 FOR Y=220 TO 400:MOVE 0,Y:DRAWR 640,0,2:NEXT
Y
3070 FOR Y=100 TO 180:MOVE 250,Y:DRAWR 120,0,11:NE
XT:FOR Y=100 TO 130:MOVE 300,Y:DRAWR 20,0,5:NEXT
3080 FOR Y=180 TO 210:MOVE 310,Y:DRAWR 270-Y,0,3:M
```

```
OVE 310,Y:DRAWR Y-270,0,3:NEXT Y
3090 X%=500:Y%=350:COL%=1:GOSUB 3890
3100 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 13. Outside the church

```
3110 REM ** location 13 **
3120 INK 13,9:PAPER 13:CLS:FOR Y=150 TO 400: MOVE
0,Y:DRAWR 640,0,2:NEXT Y
3130 X%=450:Y%=350:COL%=1:GOSUB 3890:FOR Y=150 TO
300:MOVE 70,Y:DRAWR 320,0,9:NEXT Y
3140 X%=135:Y%=190:COL%=3:GOSUB 3890:FOR Y=150 TO
210:MOVE 135,Y:DRAWR 40,0,3:NEXT
3150 FOR Y=150 TO 350:MOVE 60,Y:DRAWR 40,0,8:NEXT
Y:X%=60:Y%=330:COL%=8:GOSUB 3890
3160 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 14. In the interrogation room

```
3170 REM ** location 14 **
3180 PAPER 5:CLS:X%=280:Y%=300:col%=4:GOSUB 3890
3190 FOR X=297 TO 300:MOVE X,300:DRAW x,400,4:NEXT
 X
3200 FOR Y=210 TO 214:MOVE 250,Y:DRAWR 180,0,4:NEX
T Y
3210 FOR X=270 TO 274:MOVE X,150:DRAWR 0,70,4:MOVE
 X+140,150:DRAWR 0,70,4:NEXT X
3220 FOR X=120 TO 124:MOVE X,150:DRAWR 0,100,4:MOV
E X+70,150:DRAWR 0,50,4:NEXT X
3230 FOR Y=200 TO 204:MOVE 120,Y:DRAWR 70,0,4:NEXT
 Y
3240 GOSUB 2300:PAPER 0:RETURN
```

Location 15. On a grassy hillside

```
3250 REM ** location 15 **
3260 PAPER 2:CLS:INK 13,9:FOR Y=0 TO 150:MOVE Y,0:
DRAW 0,Y,12:NEXT Y

3270 FOR Y=0 TO 150:MOVE 640-Y,0:DRAW 640,Y,12:NEX
T Y
3280 DEG:FOR R=45 TO 140 STEP 0.4:MOVE 320,-350:DR

AWR 400*COS(R),500*SIN(R),13:NEXT R
3290 X%=550:Y%=350:col%=1:GOSUB 3890
3300 FOR X=280 TO 300:MOVE X,140:DRAWR 0,10,3:NEXT
 X
3310 FOR X=276 TO 305:MOVE X,151:DRAWR 0,1,0:NEXT
X
3320 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 16. Outside the house

```
3330 REM ** location 16 **
3340 PAPER 2:CLS:FOR Y=1 TO 100:MOVE 0,Y:DRAWR 640
,0,5:NEXT Y
3350 FOR Y=101 TO 251:MOVE 50,Y:DRAWR 200,0,9:NEXT
 Y
3360 FOR Y=240 TO 352:MOVE Y-200,Y:DRAWR (352-Y)*2
,0,3:NEXT
3370 FOR Y=290 TO 320:MOVE 100,Y:DRAWR 10,0,0:NEXT
 Y
3380 FOR Y=120 TO 150: MOVE 95,Y:DRAWR 40,0,10:MOV
E 95,Y+70:DRAWR 40,0,10:NEXT Y
3390 FOR Y=102 TO 162:MOVE 180,Y:DRAWR 20,0,12:NEX
T Y
3400 X%=550:Y%=350:col%=4:GOSUB 3890
3410 GOSUB 2300:PAPER 0:RETURN
```

Location 17. At the end of the rainbow

```
3420 REM ** location 17 **
3430 PAPER 5:CLS:INK 13,9::FOR Y=1 TO 150:MOVE 0,Y
:DRAWR 640,0,13:NEXT Y
3440 DEG:FOR Y=0 TO 13:FOR R=0 TO 180:MOVE 320,151
3450 PLOT 320+(190+Y*3)*COS(R),151+(190+Y*3)*SIN(R
),Y
3460 NEXT R,Y
3470 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 18. Next to the strange wall

```
3480 REM ** location 18 **
3490 PAPER 2:CLS:INK 13,9:FOR Y=1 TO 150:MOVE 0,Y:
DRAWR 640,0,13:NEXT Y:FOR Y=151 TO 350:MOVE 0,Y:DR
AWR 640,0,5:NEXT Y
3500 COL%=0:FOR X%=20 TO 600 STEP 40:FOR Y%=180 TO
 320 STEP 40:COL%=COL%+1:IF COL%=5 THEN COL%=6
3510 IF COL%>13 THEN COL%=0
3520 GOSUB 3890: NEXT Y%,X%
3530 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 19. Outside the yellow building

```
3540 REM ** location 19 **
3550 PAPER 2:CLS:FOR Y=1 TO 100:MOVE 0,Y:DRAWR 640
,0,5:NEXT Y
3560 FOR Y=101 TO 320:MOVE 1,Y:DRAWR 400,0,1:NEXT
Y:FOR Y=321 TO 400: MOVE 1,Y:DRAWR 740-Y,0,3:NEXT
Y
3570 FOR Y=101 TO 160:MOVE 401,Y:DRAW 640,Y,9:NEXT
 Y:FOR Y=161 TO 171:MOVE 401,Y:DRAW 640,Y,5:NEXT Y
```

```
3580 FOR Y=101 TO 150:MOVE 440,Y:DRAW 590,Y,6:NEXT
 Y:FOR Y=101 TO 150:MOVE 290,Y:DRAWR 30,0,6:NEXT Y
3590 FOR Y=120 TO 140:MOVE 40,Y:DRAWR 60,0,7:MOVE
140,Y:DRAWR 60,0,7:NEXT Y
3600 FOR Z=1 TO 3:FOR Y=120 TO 140:MOVE 40,Y+Z*50:
DRAWR 60,0,7:MOVE 140,Y+Z*50:DRAWR 60,0,7:MOVE 240
,Y+Z*50:DRAWR 60,0,7:NEXT Y,Z
3610 GOSUB 2300:PAPER 0:RETURN
```

Location 20. On a grassy plain

```
3620 REM ** location 20 **
3630 PAPER 0:CLS:INK 13,9:FOR Y=1 TO 200:MOVE 0,Y:
DRAWR 640,0,13:NEXT Y:FOR Y=1 TO 200:MOVE 250+Y/4,
Y:DRAWR 200-y/2,0,9:NEXT Y
3640 FOR Y=201 TO 390:MOVE 10+Y,Y:DRAWR 800-Y*2,0,
12:NEXT Y
3650 X%=50:Y%=350:COL%=4:GOSUB 3890
3660 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 21. At the entrance to a gloomy cavern

```
3670 REM ** location 21 **
3680 PAPER 5:INK 13,9:CLS:FOR Y=1 TO 100:MOVE 0,Y:
DRAWR 640,0,13:NEXT Y
3690 DEG:FOR R=0 TO 180 STEP 0.4:MOVE 320,100:DRAW
R 200*COS(R),300*SIN(R),8:NEXT R
3700 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 22. In the ghost's cavern

```
3710 REM ** location 22 **
3720 PAPER 5:CLS:FOR Y=1 TO 20 :FOR X=1 TO 20:LOCA
TE X,Y:PRINT ghost$;:LOCATE X,Y:PRINT er$;:NEXT X,
Y
3730 LOCATE 10,5:PRINT ghost$:GOSUB 2300:PAPER 0:R
ETURN
```

Location 23. In the cavern of pyramids

```
3740 REM ** location 23 **
3750 PAPER 5:CLS:INK 13,9:FOR Y=1 TO 70:MOVE 0,Y:D

RAWR 640,0,13:NEXT Y
3760 FOR Y=70 TO 320: MOVE Y,Y:DRAW 640-Y,Y,4:NEXT
 Y

3770 X%=40:Y%=340:COL%=3:GOSUB 3890
3780 GOSUB 2300:PAPER 0:INK 13,22:RETURN
```

Location 24. Outside a large office block

```
3790 REM ** location 24 **
3800 PAPER 2:CLS:FOR Y=1 TO 100:MOVE 0,Y:DRAWR 640
,0,12:NEXT Y
3810 FOR Y=101 TO 320:MOVE 1,Y:DRAWR 400,0,4:NEXT
Y:FOR Y=321 TO 400: MOVE 1,Y:DRAWR 740-Y,0,3:NEXT
Y
3820 FOR Y=101 TO 160:MOVE 401,Y:DRAW 640,Y,7:NEXT
 Y:FOR Y=161 TO 171:MOVE 401,Y:DRAW 640,Y,5:NEXT Y
3830 FOR Y=101 TO 150:MOVE 440,Y:DRAW 590,Y,5:NEXT
 Y:FOR Y=101 TO 150:MOVE 290,Y:DRAWR 30,0,7:NEXT Y
3840 FOR Y=120 TO 140:MOVE 40,Y:DRAWR 60,0,8:MOVE
140,Y:DRAWR 60,0,8:NEXT Y
3850 FOR Z=1 TO 3:FOR Y=120 TO 140:MOVE 40,Y+Z*50:
DRAWR 60,0,8:MOVE 140,Y+Z*50:DRAWR 60,0,8:MOVE 240
,Y+Z*50:DRAWR 60,0,8:NEXT Y,Z
3860 GOSUB 2300:PAPER 0:RETURN
3870 END
```

The only place where any animation is included in the graphics is in location 22. This is in the cavern, where the evil ghost prevents you from moving further into the cave system.

The picture of the ghost is held in the variable ghost$ and this is erased from the screen by the variable er$. Both these variables are defined at the start of the program and this will be discussed later. Line 3720 is used to move the ghost across the screen and this line may be simplified if movement is not required. Notice that the moving graphics are displayed each time location 22 is entered, even if the ghost has been killed. You may like to try displaying the graphics for a dead ghost !

At the end of each subroutine, the player must press the space bar before returning to the text part of the game. To achieve this, the program calls the subroutine at line 2300. Before returning to the text screen, the paper has to be set back to its default value by the PAPER 0 command and the colours of any INKs which have been changed must also be set back to their original setting.

In a few locations, large circles are drawn using the SIN and COS functions, but in a number of locations, small circles are drawn. In order to draw these very quickly, the coordinates are looked up rather than calculated and I have written a short subroutine to do this.

Drawing circles

```
3880 REM ** draw circle **
3890 RESTORE 3910:MOVE X%,Y%:FOR X=1 TO 40:READ D
3900 MOVE X%+(40-D)/2,X+Y%:DRAWR D,0,COL%:NEXT X:R
ETURN
3910 DATA 6,10,16,20,24,26,28,30,32,33,34,35,36,37
,38,38,39,39,39,40,41,40,39,39,39,38,37,36,35,34,3
3,32,30,28,26,24,20,16,10,6
```

Line

3890	restore the data pointer, move to the X and Y coordinates and read the data.
3900	draw the 40 lines for each circle and return.
3910	data for the circle.

Using a look up table to draw a circle makes it a little quicker than forcing the computer to calculate the coordinates. Before this routine is called, the value of X%, Y% and COL% must be set to the x and y coordinates plus the number of the pen for the colour. The most time consuming part of the whole process of creating a graphical adventure is, without doubt, the graphic design stage. At this stage, you are probably eager to try out the program and therefore I've left detailed explanations of graphics techniques to Chapter 18, where they can be given a much fuller treatment.

There are, however, a number of other approaches to graphics which can be adopted when writing a graphics adventure. In this program, I have stored the picture for each location within a different subroutine. This is a method which has the advantage of drawing the pictures very quickly, but suffers the major disadvantage of using vast quantities of memory. Other programmers prefer to store the data for their graphics within DATA lines and READ the coordinates as they are drawn. This approach is often far more efficient in memory usage, but it does take far longer to develop the program and, in addition, it can also reduce the speed with which the pictures are drawn.

A totally different approach can be adopted if you have a disc drive. The pictures for each location can be created using one of the excellent graphics packages, such as the one marketed by AMSOFT and the screen picture can be stored as a file on the disc. Thus a single disc can store the pictures for each location in the game and whenever the player enters a new location, the picture can be downloaded into memory. Using this method of graphics overlays is very efficient in memory usage. No longer do we have to store the information for the graphics within the program, leaving far more space for puzzles and locations. Unfortunately, however, this approach is totally unsuitable for a tape based game. With a disc drive it is possible to search for, and load a file containing the graphics within seconds, whilst the same process from tape would take several minutes. You would be constantly using fast forward and rewind on your recorder and would soon lose patience with the game.

Snow White: part 2 10

Once upon a time there was a young princess whose skin was white as snow, whose cheeks were as red as roses and whose hair was black as ebony. She was called Snow White and she lived with her stepmother who was beautiful and very vain. Each morning, she would look into her magic mirror and ask, 'Mirror, mirror, on the wall, who is the fairest of us all ?'. One day, instead of the usual reply, the mirror replied :– 'Queen thou art fairest in this hall, but Snow White is the fairest of us all'. The Queen was so angry that she called a servant and ordered him to take Snow White to the forest and kill her. The servant could not bring himself to do this dreadful deed, and instead left Snow White to fend for herself in the forest. After many hours, Snow White stumbled across a small cottage, where she found seven dwarves. They took her in and looked after her. Imagine the Queen's surprise when she asked the usual question of her magic mirror and got the reply that Snow White was still alive. She knew that her servant had deceived her and this made her very angry. Next morning, she set off for the cottage in the woods disguised as a poor beggar woman. When she arrived at the cottage, she knocked on the door and gave Snow White a poisoned apple.

That evening when the dwarves arrived home and found Snow White on the floor, they took her and laid her in a crystal case in a forest glade, where she lies to this very day. You are the handsome prince who has set out on a perilous journey to revive the beautiful princess.

After that piece of scene setting, let's get on with the programming. Before incorporating the graphics section within the main game, we must first type in the data lines containing the descriptions of the locations, the objects and the words understood. You would be well advised to write the graphics section using very large line numbers so as to leave plenty of space for the main program. The program can always be renumbered at a later stage to make it easier for others to type in.

Initialising the program

```
10 REM ** Snow White **
20 SYMBOL AFTER 230
30 SYMBOL 231,1,1,3,63,115,227,227
40 SYMBOL 232,255,254,252,247,226,224,224,126
50 SYMBOL 233,31,128,128,192,252,206,199,199
60 SYMBOL 234,255,127,63,247,163,3,23,190
70 ghost$=CHR$(231)+CHR$(233)+CHR$(10)+CHR$(8)+CHR
$(8)+CHR$(232)+CHR$(234)
80 er$=CHR$(32)+CHR$(32)+CHR$(10)+CHR$(8)+CHR$(8)+
CHR$(32)+CHR$(32)
90 MODE 1:CALL &BC02:BORDER 22:LOCATE 14,2:PRINT"S
now White"
100 LOCATE 2,10:PRINT"An Adventure  for the Amstra
d Computer"
110 LOCATE 12,20:PRINT"by Steve W. Lucas"
120 DIM S%(24,4),Q$(24),B$(24),B%(24),N$(24),N%(24
),V$(4),A(24)
130 RESTORE:FOR X=1 TO 24:READ Q$(X):FOR Y=1 TO 4:
READ S%(X,Y):NEXT Y,X
140 REM ** DATA FOR LOCATIONS **
150 DATA outside a small house. The door is open.A
 footpath leads to the west.,0,0,0,2
160 DATA on a wide road. A narrow footpath leads e
ast to a small house.,4,12,1,0
170 DATA in a small room. There's not much       f
urniture in here.,0,0,0,0
180 DATA in the misty mountains. There is a wide r
oad to the south and a narrow footpath leads east.
,0,2,5,0
190 DATA outside the cavern of light. The path    l
eads straight into the cave.,0,0,0,4
200 DATA in a narrow corridor. To the south lie  t
he mountains and I can see light from  the north.,
7,5,0,0
210 DATA on the sea shore. A small yacht is       m
oored here. The cavern of light lies   to the sout
h.,0,6,0,0
220 DATA in the cabin of the yacht. I can see a   s
mall island in the distance.,0,0,0,0
230 A(R)=0
240 DATA on a small island. The yacht is moored  h
ere. There is a large viaduct in front of me.,0,10
,0,0
250 DATA on a high mountain pass. The path is     b-
locked by a fall of rubble.,9,0,0,0
260 DATA in a strange room inside the old church.T
he walls are painted red.,0,14,0,0
270 DATA in a field of ripening corn. There is a b
rightly coloured building in the        distance.,2
,15,0,13
280 DATA outside the ruins of an old church. An   o
pen doorway leads into the ruins.,0,0,12,0
290 DATA in an interrogation room. A small table s
```

```
tands underneath a bright light and a  chair stand
s at one side.,11,0,0,0
300 DATA on a grassy hillside. There is a small  b
uilding in the distance.,12,17,16,0
310 DATA outside a small house. The door is      l
ocked at the moment.,0,0,0,15
320 DATA at the end of the rainbow. A wide road  l
eads through the rays of light.,15,18,0,0
330 DATA next to a strange wall. It is painted   w
ith brightly coloured circles.,17,20,19,0
340 DATA outside a large yellow building. A larged
og stands by the door.,0,0,24,18
350 DATA on a grassy plain. A green pyramid      s
tands in the distance.,18,0,0,21
360 DATA at the entrance to a large gloomy cavern,
0,22,20,0
370 DATA in a dark cavern. An evil ghost prevents
me moving further into the cavern.,21,0,0,0
380 DATA in a large cavern. An enormous pyramid  o
f solid ice stands at the centre.,0,0,0,22
390 DATA outside an office block. All the doors  a
re closed.,0,0,0,19
```

Line

20	allows the characters after CHR$(230) to be redefined.
30–60	redefine characters to make up graphics.
70	define the graphics for the ghost.
80	define the graphics to erase the ghost.
90	select text mode, reset the colours to their default values, print title.
100–110	titles.
120	dimension arrays.
130	read the data for the locations and map.
140–390	data for the locations and the map.

The first thing you'll notice about this listing is that I've used exactly the same variable names as in the previous game. Just to remind you what they are, I've summarised the major ones below.

S%(X,Y)	holds the map.
Q$(X)	holds the descriptions of the locations.
G$(X)	holds the descriptions of the objects.
B%(X)	holds the number of the location where the objects are found.
N$(X)	holds the names of the words recognised.
N%(X)	holds the pointer to the words recognised.
V$(X)	holds the descriptions of the objects being carried.
A(X)	flag to test if the object is being carried.
P%(X)	holds the number of the current location.
S%	holds the score.

As in previous games, each line of data in this section contains a description of a location followed by four numbers, corresponding to the number of the location reached by going north, south, east or west. You'll also need to ensure that no words are split on the screen in the descriptions of the locations. For this reason, some of the lines have extra spaces inserted between words, or even no spaces at all!

Locations in *Snow White*

1 Outside a small house. The door is open, A footpath leads to the west.
2 On a wide road. A narrow footpath leads east into a small house.
3 In a small room with very little furniture.
4 In the misty mountains. A wide road leads south and a narrow footpath leads east.
5 Outside the cavern of light. A path leads into the cave.
6 In a narrow corridor. A light can be seen to the north.
7 On the sea shore. A yacht is moored here.
8 In the cabin of the yacht. Through the window you can see a small island.
9 On a small island. The yacht is moored here and a viaduct can be seen.
10 On a high mountain path. Rubble blocks your way.
11 In a strange room inside the old church.
12 In a field of ripening corn. A brightly coloured building is to be seen.
13 Outside the ruins of the church.
14 In the interrogation room. A table and chair stand underneath a bright light.
15 On a grassy hillside. A small building can be seen in the distance.
16 Outside a small house. The door is locked.
17 At the end of the rainbow. A wide road leads straight through the centre.
18 Next to a strange wall covered in coloured circles.
19 Outside the yellow building. A large dog stands guard.
20 On a grassy plain. A green pyramid stands in the centre.
21 At the end of a large gloomy cavern.
22 In the dark cavern. The ghost blocks your way.
23 In the cavern. An enormous pyramid of ice stands in the centre.
24 Outside the large office block.

```
400 FOR X=1 TO 24:READ G$(X),B%(X),N$(X):N%(X)=X:
NEXT X
410 DATA a sail,5,sail,a rope,6,rope,a rudder,7,ru
dder
420 DATA a bowl of soup,3,soup,a wild cat,4,cat
```

```
430 DATA a golden casket,23,casket,"",23,"",a gold
en harp,10,harp
440 DATA a screwdriver,9,screwdriver,a giant lizar
d,8,lizard,a brass knocker,16,knocker,"",16,""
450 DATA a large red button,24,button,an old lady,
19,lady,a pot of gold,17,gold
460 DATA an enormous man with a gun in his hand,14
,man,"",14,""
470 DATA a pile of straw,2,straw,a wooden plank,1,
plank,a large pot hole,15,pot hole
480 DATA a ballpoint pen,20,ballpoint,a piece of p
aper,11,paper,a python,12,python,a magpie,21,magpi
e
```

In the next section of the program, the DATA for the 24 objects found within the game is READ into the arrays. Three of these objects are, as in the previous game, invisible at first and only become visible after solving some problems. These are listed in the table below.

Object number	*Location found in*	*Changes to*
7	23	Snow White
12	16	A silver sword
17	14	A key

Line

400	reads the data into the arrays.
410-470	DATA for the objects, their location and word recognised.

Objects found in *Snow White*

Number	*Object*	*Invisible at start?*	*Location*
1	A sail	No	5
2	A rope	No	6
3	A rudder	No	7
4	A bowl of soup	No	3

5	A wild cat	No	4
6	A golden casket	No	23
7	Snow White	Yes	23
8	A harp	No	10
9	A screwdriver	No	9
10	A lizard	No	8
11	A door knocker	No	16
12	A silver sword	Yes	16
13	A button	No	24
14	An old lady	No	19
15	A pot of gold	No	17
16	A man with a gun	No	14
17	A key	Yes	14
18	Straw	No	2
19	A wooden plank	No	1
20	A pot hole	No	15
21	A pen	No	20
22	Paper	No	11
23	A python	No	12
24	A magpie	No	21

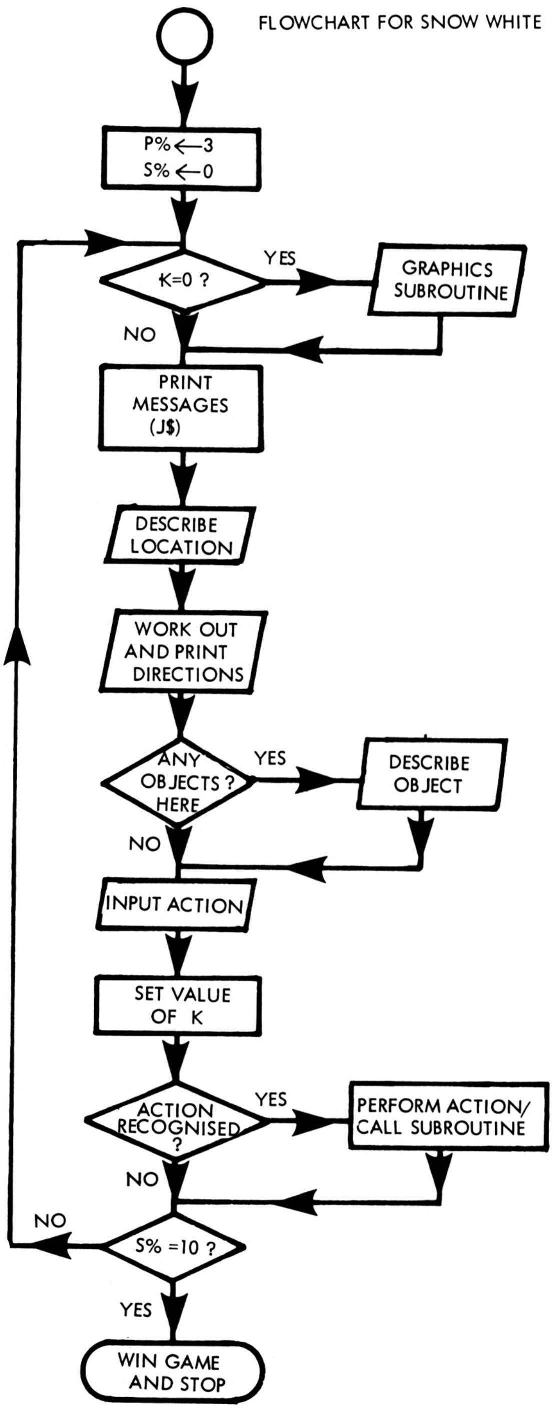
The main control loop

In a graphic adventure, it is even more important that the main control section is well structured and, once again, the best way of doing this is to use a flowchart (Fig 10.1)

The major difference between this flowchart and the one used in the previous game is that the value of the flag K is checked at the start. This flag is changed every time you move into a new location. If its value remains zero, the program is sent to the subroutine which controls the graphics and the appropriate picture is displayed on the screen.

Before examining the workings of this control section, we must look more carefully at how the graphics are introduced. The variable K is used as a flag to determine whether the graphics of the current location (P%) are to be drawn or not, and its value is checked right at the start of the loop. If K is equal to zero, then control is passed to the graphics routine and the picture corresponding to the current location is drawn (line 530 calls the graphics routine at line 2250). The next line makes sure that the computer is in the 40 column mode for text and sets the value of the flag to zero to make sure that it is always the same after drawing the graphics.

Immediately after the player types in his instructions, the value of K is changed to 1 (line 770). Its value is then changed back to zero if the player moves to a new location (lines 780-830), moves in or out of a building, or types 'look' (line 860). Thus when the score is


FLOWCHART FOR SNOW WHITE
P% ← 3
S% ← 0
K=0 ?
YES
GRAPHICS SUBROUTINE
NO
PRINT MESSAGES (J$)
DESCRIBE LOCATION
WORK OUT AND PRINT DIRECTIONS
ANY OBJECTS HERE ?
YES
DESCRIBE OBJECT
NO
INPUT ACTION
SET VALUE OF K
ACTION RECOGNISED ?
YES
PERFORM ACTION/ CALL SUBROUTINE
NO
S% =10 ?
NO
YES
WIN GAME AND STOP

checked at line 1070 and the program is sent back to the beginning of the loop, graphics will only be drawn if the player has moved location or has asked to see the picture by typing 'look'.

```
490 REM ** set the variables **
500 P%=3:S%=0
510 REM ** main control loop **
520 WHILE s%<10
530 IF K=0 THEN GOSUB 2250
540 MODE 1:K=0:PRINT J$
550 IF P%=4 THEN SH=SH+1:IF SH>3 THEN PRINT"The ca
t looks agitated!"
560 IF P%=4 AND SH>5 THEN E$="The cat attacks me!"
:GOSUB 1230
570 PRINT:PEN 1:PRINT"I am :-":PEN 2:PRINT Q$(P%)
580 REM ** describe directions **
590 A$="":IF S%(P%,1)>0 THEN A$="North"
600 IF S%(P%,2)>0 AND LEN(A$)>0 THEN A$=A$+",South
" ELSE IF S%(P%,2)>0 THEN A$="South"
610 IF S%(P%,3)>0 AND LEN(A$)>0 THEN A$=A$+",East"
 ELSE IF S%(P%,3)>0 THEN A$="East"
620 IF S%(P%,4)>0 AND LEN(A$)>0 THEN A$=A$+",West"
 ELSE IF S%(P%,4)>0 THEN A$="West"
630 IF P%=3 THEN A$="Out" ELSE IF P%=11 THEN A$=A$
+",Out"
640 IF P%=13 OR P%=5 OR P%=16 OR P%=1 OR P%=19 OR
P%=24 THEN A$=A$+",In"
650 IF A$="" THEN A$="nowhere obvious!"
660 PEN 1:PRINT:PRINT"I can go :-":PEN 2:PRINT A$:
PRINT
670 REM ** describe objects **
680 E=0:FOR T=1 TO 24
690 P=0:IF B%(T)=P% THEN P=1
700 IF P=1 THEN 720
710 NEXT T:GOTO 740
720 IF E=0 THEN PEN 1:PRINT"I can see :-":PEN 2
730 PRINT G$(T):E=1:GOTO 710
740 PRINT:PEN 1:INPUT"What shall I do now ";Z$
750 REM ** analyse input **
760 Z$=LOWER$(Z$):B$=LEFT$(Z$,2):C$=LEFT$(Z$,3):D$
=LEFT$(Z$,4):J$=""
770 PRINT CHR$(7):CLS:K=1
780 IF (B$="n" OR D$="go n") AND S%(P%,1)>0 THEN P
%=S%(P%,1):K=0 ELSE IF (B$="n" OR D$="go n") THEN
J$="I can't go that way!"
790 IF p%=15 AND (B$="s" OR D$="go s") AND SC=0 TH
EN E$="I fall down the hole and die!":GOSUB 1230
800 IF (B$="s" OR D$="go s") AND S%(P%,2)>0 THEN P
%=S%(P%,2):K=0 ELSE IF (B$="s" OR D$="go s") THEN
J$="I can't go that way!"
810 IF (B$="e" OR D$="go e") AND S%(P%,3)>0 THEN P
%=S%(P%,3):K=0 ELSE IF (B$="e" OR D$="go e") THEN
J$="I can't go that way!"
820 IF P%=19 AND SE=0 AND (B$="w" OR D$="go w") TH
```

```
EN J$="The old lady won't let me pass !":GOTO 530
830 IF (B$="w" OR D$="go w") AND S%(P%,4)>0 THEN P
%=S%(P%,4):K=0 ELSE IF (B$="w" OR D$="go w") THEN
J$="I can't go that way!"
840 IF C$="out" OR D$="go o" THEN GOSUB 3930
850 IF C$="in" OR D$="go i" THEN GOSUB 1140
860 IF C$="loo" OR C$="exa" THEN K=0
870 IF C$="swi" THEN GOSUB 1200
880 IF C$="get" OR C$="tak" OR C$="gra"  THEN GOSU
B 1290
890 IF C$="inv" THEN GOSUB 1500
900 IF C$="dro" OR C$="lea"  THEN GOSUB 1560
910 IF C$="unl" THEN GOSUB 1650
920 IF C$="pla" THEN GOSUB 1710
930 IF C$="pra" THEN GOSUB 1770
940 IF C$="kno" THEN GOSUB 1820
950 IF C$="pre" OR C$="rin" THEN GOSUB 1920
960 IF C$="rea" THEN GOSUB 1970
970 IF C$="giv" THEN GOSUB 2000
980 IF C$="sta" OR C$="kil" OR C$="use" THEN GOSUB
 2080
990 IF C$="dri" THEN GOSUB 2130
1000 IF C$="sco" THEN J$="This is no game you know
 !!!!"
1010 IF C$="hel" THEN J$="I'm sorry I don't have a
 clue!"
1020 IF C$="sea" THEN J$="I didn't find anything!"
1030 IF C$="row" OR C$="sai" OR D$="go b" THEN GOS
UB 2160
1040 IF C$="lan" OR C$="dis" OR D$="go l" THEN GOS
UB 2210
1050 IF C$="kis" THEN GOSUB 3970
1060 REM ** if score is less than 10, jump back ag
ain **
1070 WEND
```

In the previous game, the mode was not changed during play and we were able to print any message directly onto the screen. In this game, however, the mode is changed between MODE 0 and MODE 1, which would effectively clear away any message printed on the screen. The easiest way to solve this problem proved to be by using the variable J$ to hold any messages. To illustrate this, consider what happens if the player types in 'help' when asked 'What shall I do now?' in line 740. The program will compare this response with the contents of C$ in lines 780-1050 and will find a match in line 1010. It then sets the contents of the variable J$ to hold the message 'I'm sorry I don't have a clue!', so that this can be printed immediately after the screen has been cleared by the MODE 1 command in line 540.

Line	
500	sets the starting position to location 3 and the score to zero.
520-1070	main WHILE-WEND loop to repeat the loop until the score reaches 10.
530	if the value of the flag K is zero, call the graphics control section.
540	change to the 40 column mode, set the flag K and print any message held in J$.
550	test to see if the player is in location 4 and increase the value of the flag SH. If the flag is greater than 3, print the message.
560	test the location again to see if the player is in location 4 and if the flag is greater than 5 lose the game.
570	print the description of the current location (P%).
580-620	test to see if movement north, south, east or west is possible and store the information in A$.
630	check to see if the player can go out, and store information in A$.
640	check to see if it is possible to go in, and store this in A$.
650	if A$ is still empty, store the message in A$.
660	print the description of the directions possible.
670-710	examine all 24 objects to see if they are in the current location (P%).
720	if it is the first object in the location, print the message 'I can see :-'.
730	print the description of objects that can be seen and jump back to line 710.
740	input the player's instructions into Z$.
760	convert the instructions into lower case and store the first few letters in B$, C$ and D$.
770	short sound to make sure that the player knows that he has pressed ‹ENTER›, clear screen and set K = 1 to suppress any graphics.
780	deal with movement north.
790	if the player tries to move south from location 15 without having dropped the plank first, he loses the game.
800	deals with movement south.
810	deals with movement east.
820	if the player tries to move west from location 19 without first giving the lady the pot of gold, she won't let him.
830	deals with movement west.
840	deals with movement out of a location.
850	deals with movement into a location.

860	if the player wants to 'look' or 'examine', the flag is set to zero to allow graphics to be drawn.
870	deals with swimming.
880	calls the subroutine to get an object if the player types 'get', 'take' or 'grab'.
890	calls the 'inventory' subroutine to see which items are being carried.
900	call the subroutine to 'drop' an item being carried.
910	call subroutine to deal with unlocking something.
920	call subroutine to play the harp.
930	pray ?
940	knock ?
950	press or ring the bell ?
960	read ?
970	give the pot of gold to the old lady ?
980	stab, kill or use a weapon ?
990	drink ?
1000	want the score ?
1010	want help ?
1020	search ?
1030	row, sail, or go boat ? This will take you onto the yacht if you are in the correct place.
1040	land, disembark or go land ? This will allow you to get off the boat.
1050	kiss ?
1070	test the score and finish the loop.

In the previous game, all the puzzles were written inside subroutines; in order to show you that this is not the only way to set problems in an adventure, I have included three puzzles within the main control loop of this game. Every move spent by the player in location 4, increases the value of the flag SH by one (line 550). The fifth object in this game, the wild cat, is to be found in location 4. After three moves spent in this location, the value of SH reaches 3 and the computer prints the message that 'the cat looks agitated' (line 550). When the player has spent five moves in this location, the cat decides to attack and the message is stored in the variable E$ before the lose game subroutine is called in line 560. This is identical to the way in which the death routine worked in the previous game.

The second puzzle is set in line 790, which tests to see if the player tries to move south from location 15 without putting the plank across the pothole first. When this plank is dropped in the correct place, the flag SC is set to one, which allows movement south to occur. In order to solve this puzzle, the player would have to have read the description of the location very carefully.

The final puzzle in the main control loop is set in line 820, where the player is unable to move west from location 19 unless the value of the flag SE has been set to one. If the player has not solved the problem of what the old lady wants, he could well be stuck there forever! You might like to add an extra line to allow them to 'quit' ! This should be a very easy modification to make.

WIN GAME

The WHILE WEND loop tests the value of the score (S%) and unless the value of the score is 10, the loop is repeated when the WEND statement is reached in line 1070. Few other computers have WHILE WEND loops available and in listings for most other computers, you would find line 1070 replaced with

```
1070 IF S%<10 THEN GOTO 530
```

and line 520 would be deleted. It is better, however, to use the structures available to you and if this results in the listing being easier to solve, you may like to make life more difficult for the cheat by shifting the DATA along the ASCII scale.

```
1080 REM ** win game **
1090 CLS: LOCATE 12,2:PRINT"W e l l   D o n e !"
1100 PEN 2:PRINT"You have found Snow White and aft
er a   kiss from you she awakes and you both   liv
e happily ever after."
1110 REM ** add music here if required !"
1120 END
```

Line

1090	clear the screen and print the message.
1100	change the colour and print the message about what happens to Snow White.
1110	this REM statement can be replaced by some sound effects to suit your own requirements.
1120	end the game.

I have not included a sound effect for this routine, so that you can experiment to suit your own requirements.

In this adventure, the player does not really get a true score. The player either wins or loses the game and a message to this effect is printed if he asks for the score. For this reason, the variable S% is used as a flag rather than a measure of the true score.

11 Snow White: part 3

Many of the subroutines used in this game are very similar to those already used, with only minor changes to deal with the different circumstances. It is worth while comparing these routines with their equivalent in *The Wizard's Quest* to gain some insight into how to adapt them to your own purposes. A few of the subroutines, however, have had to be written specifically for this game because of its totally different plot. One major difference between routines in this program and those already introduced is that any messages which are to be printed on the screen must be stored in the variable J$, for reasons already discussed.

GO IN

```
1130 REM ** go in **
1140 IF P%=1 THEN J$="O.K.":P%=3:K=0:RETURN
1150 IF P%=5 THEN J$="O.K.":P%=6:K=0:RETURN
1160 IF P%=13 THEN J$="O.K.":P%=11:K=0:RETURN
1170 IF P%=16 OR P%=19 OR P%=24 THEN J$="The door
s locked!":RETURN
1180 J$="Don't be absurd!":RETURN
```

This subroutine is very similar to the subroutine in *The Wizard's Quest.* There are just three locations where you can actually move to a different place by typing the command 'go in', although you may well be tempted to try going into a further three locations without success. These are summarised in the chart below.

Location	*Go in possible?*	*New location*
1 Outside house	Yes	3 Inside room
5 Outside cavern	Yes	6 In corridor
13 Outside old church	Yes	11 In church

16	Outside house	No	——
19	Outside building	No	——
24	Outside offices	No	——

Line

1140	if in location 1, set message, move to location 3, set the flag to zero to allow graphics to be drawn and return to the main control loop.
1150	if in location 5, set message, move to location 6, set the flag and return.
1160	if in location 13, set message, move to location 11, set the flag and return.
1170	if in location 16, 19 or 24, set the message to tell the player that the door is locked and return.
1180	set the message that the action is impossible and return.

In lines 1140-1160, you will notice that the variable J$, which holds any messages, is set to hold the message 'O.K.' and the variable K is set to zero before returning control to the main loop. Setting the value of K to zero has the effect of forcing the computer to display the graphics for the new location. If the player is not in any of the locations tested for in line 1170, the program will reach line 1180 and a message will be printed to indicate to the player that he is being stupid.

SWIM

```
1190 REM ** swim **
1200 IF P%=7 OR P%=8 THEN E$="I drown!":GOSUB 1230
1210 J$="Don't be silly!":RETURN
```

In this program, there are two locations (7 and 8) where the player may be tempted to go for a swim. When writing the program, I decided not to use the swim routine as part of the solution to the game and therefore anyone foolish enough to try swimming in either location will drown. Thus the variable E$ is set to hold the message 'I drown' and control is passed to the 'death' routine if the player attempts to swim in either of these places. All the other locations are on dry land and therefore when line 1210 is reached, J$ is set to hold the message 'Don't be silly!' before returning to the main loop.

Line

1200	if in location 7 or location 8, the message about drowning is stored in E$ and the lose game subroutine is called.

1210 the message 'Don't be silly' is stored in J$ and control is returned to the main loop.

Death routine

```
1220 REM ** death routine **
1230 CLS:LOCATE 1,2:PRINT E$
1240 PEN 3::LOCATE 1,10:PRINT"I am DEAD !"
1250 PEN 2:LOCATE 2,20:PRINT"Press the <SPACE BAR>
 to play again."
1260 A$="":WHILE A$<>" ":A$=INKEY$:WEND
1270 RUN
```

Line

1230 clears the screen and prints the contents of E$, which should hold a description of the cause of death.
1240 change the colour of the text and print the message.
1250 change the colour, move the cursor and print the message to press the space bar for another game.
1260 wait for the space bar to be pressed.
1270 RUN the game again.

This routine is called whenever the player falls into a trap and loses the game. Before it is called, a description of the reason for losing must be stored in the variable E$. Note that in this game, death is the only method of losing!

GET

```
1280 REM ** get objects **
1290 GOSUB 1440:IF L%<1 THEN RETURN
1300 E%=0:FOR X=1 TO 24:IF B%(X)=P% AND N%(R)=X TH
EN E%=1
1310 NEXT:IF E%=0 THEN RETURN
1320 IF R=5 OR R=10 OR R=23 THEN E$="It bites me..
.AAggghhhh!":GOSUB 1230
1330 IF R=16 THEN E$="The man shoots me as I appro
ach!":GOSUB 1230
1340 IF R=11 OR R=13 OR R=14 THEN J$="Don't be abs
urd!":RETURN
1350 IF R=24 THEN J$="The magpie keeps pecking me!
"
1360 IF R=6 THEN J$="It's too heavy!":RETURN
1370 IF R=7 THEN J$="I can't lift her!":RETURN
1380 A(R)=1
1390 E%=0:FOR X=1 TO 4
1400 IF V$(X)="" THEN V$(X)=G$(N%(R)):E%=1:X=5
1410 NEXT:IF E%=0 THEN J$="I can't carry any more!
":RETURN
1420 B%(N%(R))=0:RETURN
```

This subroutine allows the player to pick up objects and the basic framework is exactly the same as in the previous game.

Line	
1290	call the subroutine to split the input sentence into two parts and return the number of the object in the variable R.
1300-1310	search through the current positions of all 24 objects to see if the object requested is in the current location and return to the main control loop if it isn't.
1320	call the death subroutine if the player tries to pick up object numbers 5, 10 or 23.
1330	if the player tries to pick up the man, object 16, he shoots the player.
1340	prevents the player from picking up objects numbered 11, 13 or 14.
1350	if the player tries to pick up the magpie, a message is printed on the screen to give him a clue!
1360	prevents the player from getting object number 6.
1370	prevents the player from getting object number 7.
1380	set the value of the pointer A(R) to one for the item being carried.
1390-1400	include the object's description in the array V$(X) which is used as the 'inventory'
1410	hands full ?
1420	change the pointer for the object's location to zero which makes it seem to disappear.

Apart from the changed line numbers, the only differences between this routine and its equivalent in *The Wizard's Quest* occur in lines 1320-1370. These are all checks for items which can't be carried.

Any attempt to pick up the wild cat (5), the lizard (10) or the python (23) results in death in line 1320, whilst anyone silly enough to try carrying the man with the gun will get shot in the following line. In a similar way, the player is prevented from carrying the door knocker, the button and the old lady, although he won't die in the attempt. Line 1350 was inserted as a clue! When trying to get the magpie, a message is printed on the screen to say that it keeps pecking you. If you examine the inventory, once you have typed the routine in, you will see that it is possible to carry the bird. When the magpie is dropped at the feet of the man with the gun, it will peck him and force him to run away. Lines 1360 and 1370 prevent the player from carrying the casket or Snow White.

The rest of this routine is identical to the previous one and can be considered as a 'standard routine' for use in all adventures.

Split input sentence

```
1430 REM ** check items and split sentence **
1440 L$="":XX=INSTR(Z$," "):R=0
1450 L%=0:L$=RIGHT$(Z$,(LEN(Z$)-XX))
1460 IF LEN(L$)<2 THEN RETURN
1470 FOR X=1 TO 24:IF LEFT$(N$(X),LEN(L$))=L$ THEN
 L%=1:R=X
1480 NEXT:RETURN
```

Apart from the line numbers used and the number of objects catered for, this routine is exactly the same as that used in the previous game. The number of objects has been changed in line 1470 to 24.
  Just to remind you of its purpose it takes the input sentence (Z$) and splits it into two words. The second word is held in the variable L$ and this is then compared with the description of all 24 objects to see if the second word refers to one of them. If a match is found, then the variable R is set to hold the number of the object, otherwise it will remain zero.

INVENTORY

```
1490 REM ** inventory **
1500 E=0:PEN 1:PRINT"I am carrying  :-":FOR X=1 TO
 4:IF V$(X)<>"" THEN PEN 2:PRINT V$(X):PEN 1:E=1
1510 NEXT:IF E=0 THEN PEN 2:PRINT"Not a sausage!":
PEN 1
1520 LOCATE 3,20:PRINT"Press the <SPACE BAR> to co
ntinue."
1530 A$="":WHILE A$<>" ":A$=INKEY$:WEND
1540 RETURN
```

The major difference between this routine and the one used in *The Wizard's Quest* is that an extra two lines are added (lines 1520-1530) which require the player to press the space bar before returning control to the main program. The reason for adding these two lines is that when control returns to the main program, there is a mode change which will clear any messages off the screen. Thus it was necessary to prevent return to the main loop until the player has had chance to read the descriptions of the objects carried.

DROP

```
1550 REM ** drop **
1560 GOSUB 1440:IF L%<1 THEN J$="I dont't have a "
+L$:RETURN
1570 E%=0:FOR X=1 TO 4
1580 IF V$(X)=G$(N%(R)) THEN V$(X)="":E%=1
1590 NEXT:IF E%=0 THEN J$="I'm not carrying a "+L$
:RETURN
```

```
1600 B%(N%(R))=P%
1610 A(R)=0:IF R=24 AND P%=14 THEN B%(24)=0:B%(16)
=0:J$="The magpie pecks the man so much that herun
s away leaving something on the      ground!":N$(1
7)="key":G$(17)="a large brass key"
1620 IF R=19 AND P%=15 THEN SC=1:Q$(15)=Q$(15)+" T
here's a      plank across the hole!":B%(19)=0
1630 RETURN
```

This routine is called from the main control loop whenever a player tries to 'drop' or 'leave' an object being carried.

Line

1560	calls the subroutine which splits the sentence typed in by the player into two separate words and returns the number of the object referred to.
1570-1580	search through all four elements of the array V$(X) to see if the object is being carried and remove it if it is.
1590	check to see if it is not being carried.
1600	set the pointer for the object to the current location.
1610	set the contents of the array A(R) to zero so that the computer knows that it is not being carried; check if object 24 is dropped in location 14 and solve the puzzle if it is.
1620	check if object 19 is dropped in location 15 and solve the puzzle.
1630	return to the main program control loop.

There are two puzzles in this game which are solved by dropping objects in the right place. In line 1610, a check is made to see if object number 24, the magpie, is dropped in location 24. A clue was given to the player when he tried to get the magpie that it liked pecking things! When the bird is dropped, the pointer B%(X) for objects 24 (the magpie) and 16 (the man) are set to zero so that they disappear from the description of objects seen in location 14. At the same time, the description for object 17 (G$(17)= the large brass key) and the word it is recognised by (N$(17)) are changed to make it appear as if the key is left behind when the man runs away. In addition, the message about the man is stored in the variable J$.

In line 1620, a check is made to see if the plank, object 19, is dropped in location 15. If it is, it will cover the pothole so that the player can move further into the game without dying. This is achieved by setting the flag SC =1 and then making the plank disappear fron the normal objects by changing the pointer for its position to zero (B%(19)=0). The description for location 19 is then changed so that it incorporates the message that the plank lies across the hole.

UNLOCK

```
1640 REM ** unlock **
1650 IF A(17)=0 THEN J$="What with ?":RETURN
1660 IF P%=16 OR P%=19 OR P%=24 THEN J$="The key d
oesn't fit!":RETURN
1670 IF P%<>23 THEN PRINT"How can I do that here?
What a stupid   idea!":RETURN
1680 J$="The key fits and the casket opens!"
1690 O$(7)="Snow White":N$(7)="snow white":S%=9:RE
TURN
```

In this game there is only one object which can be unlocked; the casket holding *Snow White*. Once the key has been found, however, the player may well try to unlock the doors found in locations 16,19 or 24. As you well know, but the player doesn't, there is no way into these buildings and therefore the program must tell the player that the key doesn't fit!

Line

1650	check to see if the key is being carried. If A(17) = 0 then it isn't and a message is stored in J$ to be printed after returning to the main loop.
1660	check to see if the player is in location 16,19 or 24 and if he is set the message into J$ and return to the main loop.
1670	check that the player is in location 23. If not, set J$ to hold the message and return to the main loop.
1680	set the message held in J$ to tell the player that the casket opens.
1690	set the description of object 7 (Snow White) and the word recognised (N$(7)), set the score to 9 and return to the main control loop.

Although the player is told 'This isn't a game you know!' when he types 'score' during play (line 1000), the computer does keep track of the score and should the player unlock the casket, the score is set to 9/10. You may prefer the program actually to give this score to the player; try changing line 1000 to

1000 IF C$="sco" THEN PRINT "You have scored ";S%;" out of 10."

PLAY

```
1700 REM ** play **
1710 IF A(8)=0 THEN J$="I can't do that just yet!"
:RETURN
1720 RESTORE 1730:PRINT"O.K.":FOR X=1 TO 34:READ D
:SOUND 7,D,20,7:NEXT:J$="Wasn't that good ..Eh ?"
1730 DATA 478,478,426,478,379,479,478,426,379,358,
319,478,478,426,379,358,319,478,478,426,478,319,47
8,284,478,253,478,253,284,319,358,379,426,478,506,
568
1740 IF P%=19 THEN J$="The old lady thanks me for
playing for  her and says ' To get rid of the ghos
t, you must pray in the old church'.":SA=1
1750 RETURN
```

Part of the solution to this game is to be found by playing the golden harp for the old lady, who will then tell you the secret of getting rid of the ghost so that you can enter the caves. In this game I have tried to show you how to add sound effects to the game which are an integral part of the game. It is pointless adding sounds to your program unless they form a useful purpose. Here the player has to play a tune on the harp *before* the old lady will give some assistance. I have used a fairly simple tune held as DATA in line 1730. You may like to try elaborating on the sound to make it sound more like a harp. Once the player has played the harp in location 19, the flag SA is set to 1. The value of this flag is tested at a later point in the game to make sure that you have solved the problem.

Line

1710	check to make sure that the harp is being carried. If A(8) is zero, then the variable J$ is set to hold the message 'I can't do that yet!'
1720	restore the data to line 1730, print the message 'O.K.', play the tune, and set the message in J$.
1730	holds the data for the tune.
1740	test to see if the player is in location 19 and change the message if he is. The value of the flag SA is also changed if in the correct location.
1750	return to the main program control loop.

Before moving on to the next chapter, don't forget to check out the routines you have just typed in. Use the technique adopted in the last game; when you are convinced that all is well, you should save an updated copy of the game just in case !

12 Snow White: part 4

PRAY

```
1760 REM ** pray **
1770 J$="O.K.":FOR x=1 TO 1500 STEP 100:SOUND 7,x,
20,7:NEXT
1780 IF P%<>11 THEN RETURN
1790 IF SA=0 THEN J$="Nothing happens!":RETURN
1800 J$="A voice booms out 'To kill the ghost youm
ust use the silver sword!":SB=1:RETURN
```

Line

1770	set the contents of J$ to hold the message 'O.K.' and play a short tune.
1780	if praying in any location other than the church (location 11), return to the main program control loop.
1790	if the flag SA is zero, nothing happens and control is returned to the main loop.
1800	the contents of J$ are changed to hold the new message about killing the ghost. The flag SB is set to one and the program returns to the main loop.

In this game, the player must first play the harp for the old lady, who will tell them to pray in the church. At the same time, the flag SA is set to one. The value of this flag is checked when praying in the church and unless the first problem has been solved, nothing happens!

KNOCK

```
1810 REM ** knock **
1820 IF P%<>16 THEN J$="There isn't much point in
doint that     here!":RETURN
1830 PRINT"knock knock":SOUND 1,270,10,7:FOR X=1 T
O 200:NEXT X:SOUND 1,270,10,7
1840 IF SA=0 THEN J$="Nobody answers.":RETURN
1850 FOR X=1 TO 1000:NEXT X:PRINT"I hear sombebody
```

```
 walking towards the    door.":FOR x=1 TO 1000:NEX
T x
1860 IF SD=0 THEN PRINT"A man answers the door and
 throws a     silver sword onto the floor. 'Take t
hat'he says."
1870 IF SD=0  THEN G$(12)="a silver sword":N$(12)=
"sword":LOCATE 1,20:PRINT"Press the <Space Bar> to
 continue."
1880 IF SD>0 THEN E$="The man answers the door and
 says 'What you again ?', as he hits me with a bat
on":GOSUB 1230
1890 SD=1:A$="":WHILE A$<>" ":A$=INKEY$:WEND
1900 RETURN
```

Line

1820	Check to see whether the player is trying to knock anywhere other than location 16 and set the contents of J$ to hold the message that there isn't much point, if they are.
1830	print the message and make the sound effect.
1840	check the value of the flag SA. If it is zero, nobody answers the door and control is returned to the main loop.
1850	make sound effect and print the second message.
1860	check the value of the flag SD and if it is the first time the player has knocked on the door, the man answers and drops the sword.
1870	if it is the first time the player has knocked on the door, the value of G$(12) and N$(12) are changed.
1880	if it is the second time that the player has knocked on the door, the man answers and hits him. This message is stored in E$ before the lose game subroutine is called.
1890	set the flag SD to one and wait for the space bar to be pressed.
1900	return to the main program.

After visiting the church to find the clue about killing the ghost, the player must then return to location 16 and knock on the door. Although there are many doors in this game, this is the only one with a door knocker and therefore an appropriate message is defined in line 1820 if the player tries anywhere else. The sound effects for a door knocker can be achieved in many different ways and the routine adopted illustrates the important technique of introducing a time delay between events, which gives the game a 'real time' element. In line 1830, the sound is made and a short delay is created using a simple FOR NEXT loop before the second knock is sounded. Should the player not have previously played the harp for the old lady, the value of the flag SA will remain zero and nobody will answer the door (in line 1840). If it is the first knock on the door, the flag SD will be zero and the description of object 12 will be changed to make the sword appear (line 1870). At the same time, a message is printed (line 1860) which gives the impression that the man has thrown the sword out of the door, whereas in fact it has always been there!

When the player has knocked on the door and been given the sword, the flag SD is set to 1 and should he try knocking again, the value of the flag SD will be trapped by line 1880. This will result in the death routine being called, with the message E$ holding the information that the man has hit you. Line 1890 then waits for the space bar to be pressed before returning to the main loop. The reason for this is that the messages, which have been printed on the screen, will be erased by the MODE 1 command on return to the main control loop of the program.

RING

```
1910 REM ** ring **
1920 IF P%<>24 THEN J$="I can't do that here!":RET
URN
1930 PRINT"Ding Dong ":SOUND 7,239,30:FOR X=1 TO 3
00:NEXT X:SOUND 7,478,30
1940 PRINT"I hear somebody walking towards the doo
r":FOR X=1 TO 300:NEXT X
1950 E$="'What do you want ?', says a voice from b
ehind the door. A bucket of boiling oilis thrown o
nto my head from above!":GOSUB 1230
```

There is a large red button on the door in location 24 of this game, which controls the doorbell, and the player will probably try to press it. The line in the main loop which calls this subroutine responds to the instructions 'press' or 'ring'.

Line

1920	if not in location 24, set J$ to hold an appropriate message and return to the main program.
1930	print the message and make the sound effect.
1940	print the message about somebody coming.
1950	lose the game.

This subroutine was written as a 'red herring' and is meant to put the player off the correct scent. It was written to illustrate a particular technique often used in adventure games to add realism, namely introducing a time delay. The bell is rung in line 1930 and a message, 'I hear somebody coming', is printed in the next line. This is followed by a short delay, again in line 1940 before the bucket of boiling oil is flung over you. You may like to try changing the sound effect, or even the result of pressing the bell. Beware, however, of writing too many red herrings into your game as they can waste an enormous amount of RAM.

READ

```
1960 REM ** read **
1970 IF A(22)=0 THEN J$="I have nothing to read !"
:RETURN
1980 J$="There is a simple message written on thep
aper. 'FIND THE KEY'":RETURN
```

The routine to read the writing on an object being carried is not as important in this game as in *The Wizard's Quest*, and serves only to give a clue to the player.

Line

1970	if the piece of paper is not being carried, set the variable J$ to hold an appropriate message and return to the main program.
1980	set J$ to hold the message and return to the main program.

If the player is not carrying the paper, A(22) will be zero and the message variable J$ will contain the message that he has nothing to read. If the program does reach line 1980, the player must be carrying the paper and so the player is told that he must find the key.

GIVE

```
1990 REM ** give **
2000 GOSUB 1440:IF R<>15 THEN J$="There isn't any
point!":RETURN
2010 IF P%<>19 THEN J$="There is no point in doing
 that here!":RETURN
2020 IF SA=0 THEN J$="'You haven't played the harp
 for me yet!, she says.":RETURN
2030 IF A(15)=0 THEN J$="I don't have it!":RETURN
2040 J$="The old lady takes my gift and runs aways
inging 'Somewhere over the rainbow'!"
2050 B%(14)=0:B%(15)=0:FOR X=1 TO 4:IF V$(X)=G$(15
) THEN V$(X)=""
2060 NEXT:SE=1:RETURN
```

If the player attempts to move west from location number 19 without having first given the pot of gold to the old lady, she will refuse to let him (see line 820). The flag used to check whether the pot of gold has been given is SE and its value must be greater than zero if he is to escape.

Line

2000	call the subroutine which splits the sentence into two words and if the item mentioned is not number 15, set J$ to

contain the message before returning to the main program control loop.

2010 test to see if the player is in location 19 and return to the main loop if not.

2020 test the value of SA and if it is still zero, set J$ to hold a message about playing the harp first.

2030 test to see if the player is carrying the pot of gold.

2040 set the message about the old lady going.

2050 remove the old lady and the pot of gold by setting the pointers B%(14) and B%(15) to zero and remove the pot of gold from the array V$(X).

2060 set the value of the flag SE and return to the main program.

When playing this game, you will have to be very careful that you don't enter location 19 without carrying both the harp *and* the pot of gold. The harp can be found in location 10, whilst the pot of gold is found at the end of the rainbow (where else ?). If the poor unfortunate player does venture into location 19 without these items, they will be stuck and I haven't included a routine which allows them to quit the game. This could make a short project for you to add to the game.

STAB

```
2070 REM ** stab **
2080 IF A(12)=0 THEN J$="I have no suitable weapon
!":RETURN
2090 IF P%<>22 THEN J$="Don't be so violent here!"
:RETURN
2100 J$="I kill the ghost with the silver sword!"
2110 Q$(22)=LEFT$(Q$(22),18):S%(22,3)=23:RETURN
```

This subroutine is called from the main program whenever an attempt is made to 'stab', 'kill' or 'use' an object being carried. I have written the routine in this game in such a way that the computer doesn't expect two words to be input. A slight change which could be made would be to add the extra line below

2085 GOSUB 1440:IF LEFT$(L$,4)<>"ghos" THEN J$="I can't kill the ";L$:RETURN

The effect of this would be to give the player a little more information about which objects can be killed. If, for example, you were to type 'kill cat', the computer would print 'I can't kill the cat'.

Line

2080 check to see if the sword is being carried and print the message if not.

2090 check to see if in location 22, where the ghost is to be found. Print the message and return if in the wrong room.
2100 set the contents of the message string.
2110 change the description of the location, change the map to allow progress east and return to the main program.

The most important line in this section is line 2110, where the description of the location is shortened to exclude any mention of the ghost. This is done by setting the contents of the array element Q$(22) so that it holds only the first 18 letters of the previous description. The final part of the line then changes the map so that movement east from room 22 leads to location 23, where the casket containing *Snow White* is to be found. One change you may like to consider making to this routine is to set another flag so that when the picture is drawn for this location after the ghost has been killed, the ghost is no longer displayed. This could be done by making the following changes

```
2105  SK=1

3720  IF SK=1 THEN RETURN ELSE PAPER 5:CLS:FOR Y=1 TO
20:FOR X=1 TO 20:LOCATE X,Y:PRINT ghost$;:LOCATE X,Y:
PRINT er$;:NEXT X,Y
```

DRINK

```
2120 REM ** drink **
2130 IF A(4)=0 THEN J$="I have nothing to drink!":
RETURN
2140 E$="I drink the soup and fall into a deep    s
tupor. It must be poisoned!":GOSUB 1230
```

As in the previous game, the 'drink' routine is used to lead the player to his death. The only object which can be drunk is the bowl of soup, object 4.

Line

2130 check to see if the soup is being carried and return to the main loop if it isn't.
2140 set the contents of E$ to hold a message about the cause of death and call the death subroutine.

You may like to try changing this routine so that you *must* drink the bowl of soup to give you the strength to lift the harp or the plank. This could be achieved in the following way:

1 Change line 2140 to

```
2140  J$="I feel much stronger now!":SL=1:RETURN
```

2 Add the following line to the 'GET' routine.

1345 IF (R=8 OR R=19) AND SL=0 THEN J$= "I feel too weak to lift it!   ":RETURN

The flag SL is then used to test whether the player has drunk the soup and if he tries to get the harp (object 8) or the plank of wood (object 19) without the flag being set to 1, then the message 'I feel too weak to lift it!' will be printed and control returned to the main program.

SAIL

```
2150 REM ** sail boat **
2160 IF P%<7 OR P%>9 THEN J$="Just how am I suppos
ed to do that here ?":RETURN
2170 IF P%=8 THEN J$="I'm already sailing the boat
!":RETURN
2180 P%=8:SG=SG+1:K=0:IF SG>1 THEN SG=0
2190 RETURN
```

There are two locations in this game where the player must sail the boat and these are numbered 7 and 9.

Line

2160	test to see if the location is numbered less than 7 or greater than 9 and print the message if it is.
2170	test to see if the location is number 8, where the player is already aboard the boat, and return to the main program if it is.
2180	change the location to number 8 and change the value of the flag SG before returning to the main program control loop.

Line 2180 is particularly important in this routine because the flag SG is used to determine where the player's boat lands. Each time the routine is called, the value of SG will change. In the landing routine, discussed later, the player will land in location 9 if SG has a value of 1 and in location 7 if SG is equal to zero.This routine is called whenever the player types 'row', 'sail' or 'go boat' in the main control loop.

LAND BOAT

```
2200 REM ** land the boat **
2210 IF P%<>8 THEN J$="not here!":RETURN
2220 K=0:IF SG=1 THEN P%=9 ELSE P%=7
2230 RETURN
2240 END
```

This routine is called from the main control loop whenever the player types 'land', 'disembark' or 'go land'.

Line	
2210	if the location is not number 8, print the message and return to the main program.
2220	set the flag K to zero and change the position.
2230	return to the main loop.

Line 2220 is the most important line in this routine. The value of the flag K is set to zero to allow the graphics for the new location to be drawn. The flag SG is then checked to see which location the boat lands in. Thus if the player boards the boat in location 7, it will land in location 9 and vice versa.

GO OUT

```
3920 REM ** go out **
3930 IF P%=3 THEN P%=1:K=0:J$="O.K.":RETURN
3940 IF P%=11 THEN P%=13:K=0:J$="O.K.":RETURN
3950 J$="Don't be a silly billy!":RETURN
```

This is a complementary routine to the 'go in' subroutine already described and can only work in locations 3 or 11.

Line	
3930	if the location is number 3, then move to location 1, set the flag K to zero to allow graphics to be drawn and return to the main control loop.
3940	if the location is number 11, move to location 13, set the flag K to zero and return to the main loop.
3950	set the contents of the message string and return to the main program.

If the player is not inside locations 3 or 11, the program will reach line 3950 and the message 'Don't be a silly billy!' will be stored in J$ for printing on return to the main program loop.

KISS

```
3960 REM ** kiss **
3970 IF P%<>23 THEN J$="I can't do that here!":RET
URN
3980 IF S%<>9 THEN J$="The casket's locked!":RETUR
N
3990 J$="I kiss Snow White and she awakes!":S%=10:
RETURN
```

This subroutine is needed as the final stage of the solution to the game. When you have opened the casket and found *Snow White,* you must kiss her to awaken her from her slumber.

Line

3970	if the location is wrong, print the message and return to the main program.
3980	if the score is less than 9, the casket must still be locked, so the message is printed and control is returned to the main loop.
3990	print the message, set the score and return to the main loop.

The score is used as a flag in this routine to check whether the casket has been opened. It is often convenient to check the score in a game in which points are awarded for solving particular problems, rather than using a separate flag.

You should now have typed in all the subroutines for 'Snow White' and be ready to play the game. Do remember to save a copy before running it, so that if disaster does strike, you won't lose all your hard work. Like the previous game, you would be well advised to check each routine as you type it in, rather than saving your checking until the end of the game.

Suggested improvements

There is plenty of memory free in your micro after typing this program in to add a few extra routines. There is no facility at this stage to quit the game when stuck in the location with the old lady. This can be achieved by adding a line to the main control loop such as

```
915    IF C$="qui" THEN PRINT"Goodbye. Thank you for play-
ing.":END
```

Note that in this case, you don't need to store the message in J$ for printing because the program will stop at this point rather than change the mode. The other facility missing from this program is that of saving a game. Adding this facility should make an interesting exercise. In principle, the routine is very similar to that used in the previous program and you will need to check the number of locations and objects and sort out the flags used. When you have finally finished developing the game, you may as well renumber it to make life easier for anyone having to type it in.

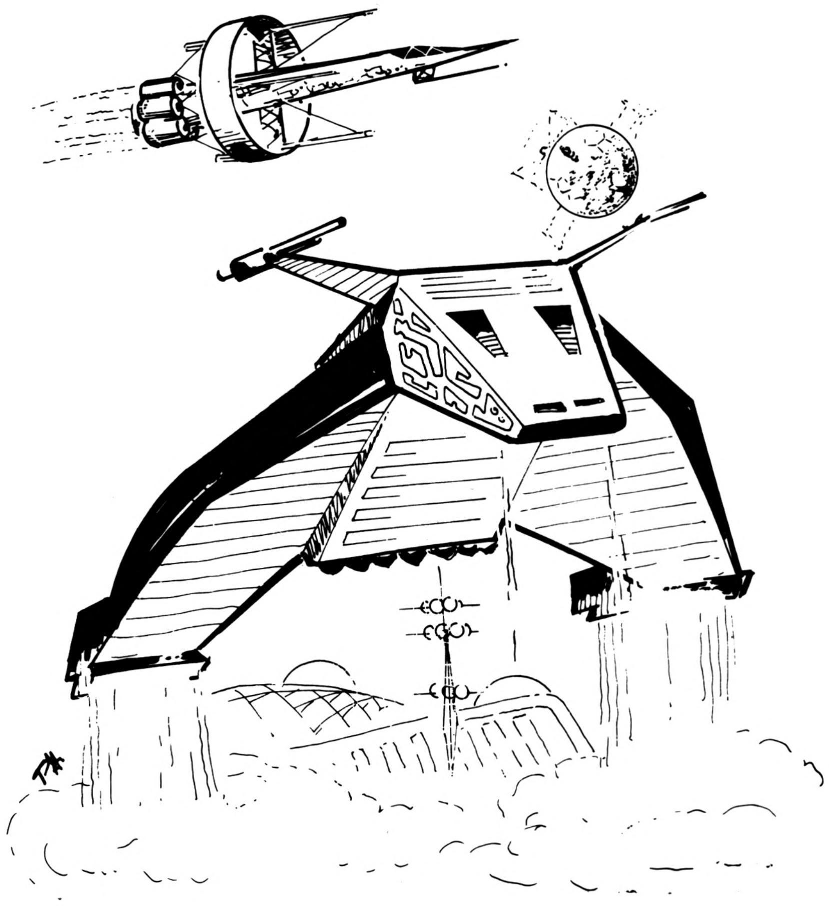
Using a data file to create an adventure 13

This adventure game loads into your computer in two parts. The first part is the main program that controls the action of the game, whilst the second part is a data file. This data file contains the descriptions of all the objects and locations found in the game, together with a list of all the words understood. Using this technique makes it very easy to create a completely new game. The program listed later on in the book which creates the first data file, makes it possible to write an adventure of your own with absolutely no knowledge of BASIC programming. All you need to do to create your masterpiece is to type in the main game and save it onto tape. You should then type in the second program and, before saving it on a different tape, you should run it. The program will ask you a series of questions and when you have answered them all, you will be asked to insert a tape into the tape recorder. This will then save the data file onto the tape — you would be well advised to save it immediately after the main program, so that you don't have to change tapes when loading the game.

The very first question you will be asked when running the file creating program is whether you want to make any changes. If you answer 'no' to this question, the file created will allow you to play *A Journey Through Space* and it is this adventure which will be explained over the next few chapters. Answering 'yes', will, of course, allow you to either make minor modifications to this game or to create a new adventure of your own.

A Journey through Space

For many years, your spaceship travelled silently through galaxies far from Earth, controlled only by a large computer. You, along with your fellow crew members, have remained in a state of suspended animation, your vital life functions being constantly monitored by the computer. Two hours ago, the computer started to awaken you from your slumber to help assess the damage caused by a meteor striking the ship. Your only course of action was to land on the nearest planet to arrange for repairs.



Unaware of the atmospheric storms of 'Lucia', you attempted to land the ship on a small platform high above the planet's surface. Unfortunately, the violent winds drove the ship off the edge of the platform. When you came round after the crash, you found that the computer had been damaged beyond repair and the life support functions had failed. You are alone and need to repair the ship so that you can return to Earth. Your task will not be easy!

Before considering the effects produced when you have created your own data file, we need to examine carefully how the main program works with the standard data file *A Journey Through Space.*

Initialising the program

```
10 REM ** ADVENTURE **
20 DIM Q$(50),S%(50,4),V$(4),G$(25),B%(25),N%(25),
N$(25),A(25)
30 REM ** main program **
40 REM ** S.W. Lucas **
50 MODE 1
60 CALL &BC02
70 GOSUB 1900
80 LOCATE 10,2:PRINT J$
90 LOCATE 2,7:PRINT"An adventure game for the Amst
rad CPC464"
100 CLS:S%=0:P%=2
```

Line	
20	dimension the arrays.
50	select the 40 column mode.
60	set the colours to their default values.
70	calls the subroutine to load the data file from tape.
80	prints the title (if it has been saved on tape!)
90	prints the message.
100	clear the screen, set the score to zero and the start location to 2.

The most important line in this section is line 70, which calls the subroutine to load the data file. Before this can be loaded, the arrays must have been dimensioned large enough to hold the information for the game. You will notice that I have used the same variable names for these arrays as before.

Immediately after the data file has been loaded from tape or disc, the title will be printed. This title may have been read off the tape, so as to allow us to change the game without having to change any of the main program. In the listing for the data file, however, I have not saved any name onto the tape and you may like to experiment with this. The game always starts off in location 2, although the position is loaded in from tape so that the same data file can be used to save the player's current position. This means that you could delete P%=2 from line 100 if you so wish.

The following list shows the objects which are found in the game produced by the standard data file, together with the location in the game where they may be found.

Objects found in the game

	Object	*Location*
1	A strong knife	1
2	A phaser	1

3	A shovel	1
4	A space suit	1
5	A button	3
6	A lever	4
7	A large can	22
8	A crystal warp control	46
9	A packet of wolf nuts	32
10	A hyper viper	17
11	A pair of leather gloves	2
12	A crystal control socket	2
13	A fuel injection cap	2
14	A damaged panel	2
15	A panel repair manual	45
16	A remote control for androids	32
17	A large hook	6
18	A boulder	6
19	A glowing statue	37
20	A lodoria plant	18
21	An alien mask	24
22	A metal bar	16
23	A fuel spout	50
24	A slot	50
25	An intergalactic credit card	33

It is worth bearing in mind that many of these objects have a specific purpose in the game and if you try to change their nature by altering the data file, you should try to keep them fairly similar in nature. As an example of this, consider the large can, object 7. In the control program, this must be taken to location number 50 to be filled with rocket fuel. This is achieved when the player inserts an intergalactic credit card into the slot in the same location. If you were to change the description of object 7 to a leopard, this would result in a completely illogical game. Imagine taking the leopard to be filled up! You could of course change it to an oil lamp, a fountain pen, an empty bottle or any other empty container which can be filled with a liquid. I shall be coming back to this point in greater detail when I introduce the program used to create the first data file.

Another point to be borne in mind when typing this program in is that you will not be able to test out each routine as it is developed in the manner adopted with the previous programs. This is because you will need to load the data file in again each time the computer comes across a mistake. You may be wondering how I actually developed this program, as it does require the data file to be created before it will work. I did, in fact, write a shortened version of the data file creator program first and, only when the main program had been fully developed, did I convert it into its final form as listed here.

The main control loop

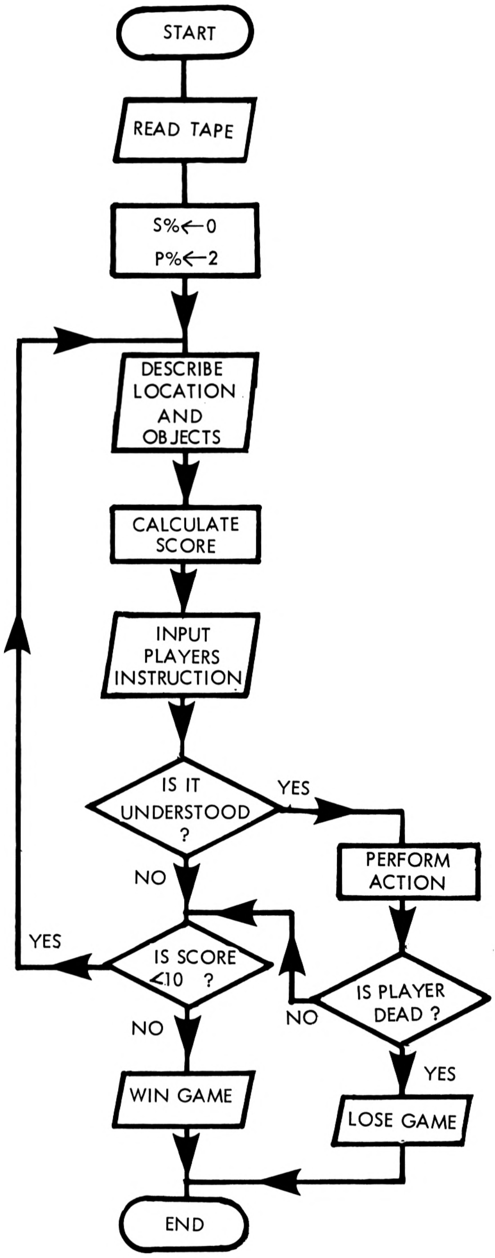


Fig. 13.1 Flowchart for control section.

The flowchart for the control section of this program is very similar to previous flowcharts. The loop is, again, repeated until the score (S%) reaches 10.

```
105 WHILE S%<10
110 PRINT"You are ":PRINT Q$(P%):PRINT
120 GOSUB 1510:REM ** check score **
130 A$="":IF S%(P%,1)>0 THEN A$="North"
140 IF S%(P%,2)>0 AND LEN(A$)>0 THEN A$=A$+",South
" ELSE IF S%(P%,2)>0 THEN A$="South"
150 IF S%(P%,3)>0 AND LEN(A$)>0 THEN A$=A$+",East"
 ELSE IF S%(P%,3)>0 THEN A$="East"
160 IF S%(P%,4)>0 AND LEN(A$)>0 THEN A$=A$+",West"
 ELSE IF S%(P%,4)>0 THEN A$="West"
170 IF P%=6 OR P%=12 THEN A$=A$+",In"
180 IF P%=9 OR P%=11 THEN A$=A$+",Out"
190 IF P%=35 THEN A$=A$+",Up"
200 IF P%=34 THEN A$=A$+",Down"
210 IF A$="" THEN A$="nowhere obvious!"
220 PRINT"You can travel ":PRINT A$
230 REM ** describe objects **
240 E=0:FOR T=1 TO 25
250 P=0:IF B%(T)=P% THEN P=1
260 IF P=1 THEN 280
270 NEXT:GOTO 300
280 IF E=0 THEN PRINT:PRINT"You can see "
290 PRINT G$(T):E=1:GOTO 270
300 PRINT:PRINT"What do you want to do now ":INPUT
 Z$
310 CLS: PRINT CHR$(7)
320 Z$=LOWER$(Z$):B$=LEFT$(Z$,2):C$=LEFT$(Z$,3):D$
=LEFT$(Z$,4)
330 IF (B$="n" OR D$="go n") AND S%(P%,1)>0 THEN P
%=S%(P%,1) ELSE IF (B$="n" OR D$="go n") THEN PRIN
T"You can't go that way!"
340 IF (B$="s" OR D$="go s") AND S%(P%,2)>0 THEN P
%=S%(P%,2) ELSE IF (B$="s" OR D$="go s") THEN PRIN
T"You can't go that way!"
350 IF (B$="e" OR D$="go e") AND S%(P%,3)>0 THEN P
%=S%(P%,3) ELSE IF (B$="e" OR D$="go e") THEN PRIN
T"You can't go that way!"
360 IF (B$="w" OR D$="go w") AND S%(P%,4)>0 THEN P
%=S%(P%,4) ELSE IF (B$="w" OR D$="go w") THEN PRIN
T"You can't go that way!"
370 IF C$="get" OR C$="tak" OR C$="gra" THEN GOSUB
 610
380 IF C$="inv" THEN GOSUB 790
390 IF C$="sco" THEN PRINT"You have scored ";S%*10
;" %"
400 IF C$="hel" THEN PRINT"Use your eyes and keep
your wits about  you!"
410 IF C$="dro" OR C$="lea" OR C$="put" THEN GOSUB
 870
420 IF C$="wea" THEN GOSUB 970
```

```
430 IF C$="in" OR D$="go i" THEN GOSUB 1020
440 IF C$="out" OR D$="go o" THEN GOSUB 1070
450 IF C$="fir" OR C$="bla" OR C$="use" THEN GOSUB
 1110
460 IF C$="dow" OR D$="go d" THEN GOSUB 1160
470 IF C$="up" OR D$="go u" THEN GOSUB 1200
480 IF C$="jum" THEN GOSUB 1240
490 IF C$="pus" OR C$="pre" THEN GOSUB 1280
500 IF C$="pul" THEN GOSUB 1380
510 IF C$="cli" THEN GOSUB 1420
520 IF C$="cut" THEN GOSUB 1470
530 IF C$="ins" THEN GOSUB 1580
540 IF C$="fil" THEN GOSUB 1700
550 IF C$="rep" OR C$="men" OR C$="fix" THEN GOSUB
 1750
560 IF C$="sav" THEN GOSUB 1750
570 IF C$="loa" THEN GOSUB 1900
580 WEND
590 CLS:LOCATE 1,10:PRINT"Well done you have solve
d this adventure":END
```

Line

110	describe the current location (P%).
120	call the subroutine to calculate the score.
130	check if movement north is possible and set the contents of A$.
140	check if movement south is possible and set the contents of A$.
150	check if movement east is possible and set the contents of A$.
160	check if movement west is possible and set the contents of A$.
170	check to see if the player is in location 6 or 12 and set A$ to allow movement 'in'.
180	check to see if the player is in location 9 or 11 and set A$ to allow movement 'out'.
190	check to see if the player is in location 35 and set A$ to allow movement 'up'.
200	check to see if the player is in location 34 and set A$ to allow movement 'down'.
210	check to see if A$ is still empty and set message to 'nowhere obvious!' if it is.
220	describe the directions in which you can travel.
240	set the flag E to zero and search through 25 objects.
250	if an object is found in the current location, set the flag P to 1.
260	if an object is found, jump to line to describe it.

270	end of loop to search through all 25 objects.
280	if the flag E is still zero, print the message 'You can see'.
290	print description of object and set the flag to one to suppress the message 'You can see' if a second object is found in the same location.

Many adventure games are written in such a way as to break the golden rule of programming and this one is no exception. In line 260, the program jumps out of a FOR NEXT loop. This is not, generally, to be recommended, although in this instance the program jumps back into the loop again when the object has been described. Despite this redeeming feature, programming purists may well like to rewrite this section of coding to adopt a better structure. As well as offending structured programming enthusiasts, jumping out of FOR NEXT loops can also cause the program to behave in an unpredictable manner. There are a number of ways of overcoming this if you do find yourself jumping out of a loop. Probably the easiest is to set the value of the control variable to one greater that the maximum value of the loop:

```
10  FOR X=1 TO 5
20  IF J=3 THEN X=6:GOTO 50
30  NEXT X
40  PRINT "end of loop"
50  PRINT "XXXXXX"
```

In this way, the value of X will be 6 whether the program jumps out of the loop or the loop is terminated in the normal manner. An alternative solution is to jump back into the loop immediately after completing the task in hand. This is not always very easy to arrange and in most cases, you would be well advised to rewrite the section of code to avoid jumping out of the loop!

Line

300	input the player's instruction.
310	clear the screen and make a short sound.
320	find the first few letters of the player's instructions and store them in B$,C$ and D$.
330	move north if possible.
340	move south if possible.
350	move east if possible.
360	move west if possible.

Lines 330 to 360 are very similar to each other and deal with movement from one place to another within the game. If the player types 'n' or 'go north', line 330 will check first of all to see if movement in that direction is possible. If S%(P%,1) is greater than 0, the value held in that location of the array corresponds to the number of the location reached by going north and the value of the current location (P%) is changed to this value. Should this value be zero, then the map of the game does not allow movement north, and the message 'You can't go that way' will be printed. The following lines check the value of S%(P%,2), S%(P%,3) and S%(P%,4) respectively to see if movement south, east or west is possible.

Lines 370 to 570 examine the first few letters of the instruction typed in by the player to see if they can be understood; if they are recognised as a valid word the appropriate subroutine is called. The words recognised in this section are: go north, n, go south, s, go east, e, go west, w, get, take, grab, inventory, score, help, drop, leave, put, wear, in, go in, out, go out, fire, blast, use, down, go down, up, go up, jump, push, press, pull, climb, cut, insert, fill, repair, mend, fix, save, load.

In some cases, where the instruction doesn't need much interpretation, it is unnecessary to call a subroutine and the action can be dealt with within the main program. If, for example, the player asks for 'help', the computer will print the message every time. The score is also dealt with in this way. Each time around the main loop, the score is calculated (line 120), so that if the player types 'score', it is only necessary for the computer to print it. In this game, the score is out of 10, although the player is given a percentage score.

If, for example, you wanted to add an extra line to the program so that the computer recognised the word 'eat', this must be inserted before line 580:

```
571 IF C$="eat" THEN PRINT "I'm not hungry at the moment
thank you!"
```

or

```
571 IF C$="eat" THEN GOSUB 2000
```

This second alternative would be necessary if you wanted to make the game more 'intelligent'.

As the game stands, responses which are not recognised by the computer are ignored. If, thus, the player types 'run', the computer will not print any message at all. This can be irritating to the player, who doesn't know whether the computer is working properly. All that needs to be done to rectify this is to use another flag. eg. K and add the following two lines :-

```
112 K=0
572 IF K= 0 THEN PRINT"I'm sorry I just don't understand you!"
```

You will, of course, need to set the value of this flag to 1 if an instruction is recognised and understood. This should be done by adding :K=1 to the end of each line from line 330 to 570:

570 IF C$="1oa" THEN GOSUB 1880:K=1

If you do decide to include this feature within the game, you will need to set K=1 in both parts of the lines dealing with movement (lines 330-360):

330 IF (B$="n" OR D$="go n") AND S%(P%,1)>0 THEN P%= S%(P%,1): K=1 ELSE IF (B$="n" OR D$="go n") THEN PRINT"You can't go that way!":K=1

If the score is less that 10 when the program reaches line 580, the loop will be repeated again. If, however, the player does manage to reach a score of 100% (S%=10), the program will leave the loop and reach line 590, where he will be told that he has won the game.

GET

```
600 REM ** get **
610 GOSUB 730: IF L%<1 THEN PRINT"You can't see a
";L$;" here!":RETURN
620 E%=0:FOR X=1 TO 25:IF B%(X)=P% AND N%(R)=X THE
N E%=1
630 NEXT:IF E%=0 THEN PRINT"You can't see a ";L$;"
 here!":RETURN
640 A(R)=1
650 IF R=10 AND A(11)<2 THEN PRINT"You need to wea
r some protection first!":RETURN
660 IF R=6 OR R=5 OR R=18 OR R=19 OR R=20 OR R=21
OR R=23 OR R=24 THEN PRINT"You can't!":RETURN
670 IF R=13 OR R=14 OR R=12 THEN PRINT"Don't be st
upid!":RETURN
680 E%=0:FOR X=1 TO 4
690 IF V$(X)="" THEN V$(X)=G$(N%(R)):E%=1:X=5
700 NEXT:IF E%=0 THEN PRINT"Your hands are full!":
RETURN
710 B%(N%(R))=0:RETURN
```

There are 11 objects in this game which cannot be picked up during play. These are listed in the chart below.

Number	*Description*	*Location found in*
6	A lever	4
5	A button	3
18	A boulder	6
19	A glowing statue	37

20	A lodoria plant	18
21	An alien mask	24
23	A fuel spout	50
24	A slot	50
13	A fuel injection cap	2
14	A damaged panel	2
12	A crystal control socket	2

In addition to these items which cannot be picked up at all, object number 10, the hyper viper, can only be 'got' when the player is wearing the leather gloves for safety. The value of A(11) is set to 2 when the player is wearing the gloves, object 11. You will need to bear this in mind if you are modifying the data file. You *must* make sure that you change the 'hyper viper' into something which can only be picked up when you are wearing some protection and you must also change the leather gloves into an object to be worn! In a similar way, the program would not seem logical if you changed the boulder into a piece of paper and were then unable to lift it!

Line	
610	call the subroutine to split the sentence into two words and store the number of any object mentioned in the variable R. Check the value of L% and if it is less than one, the object mentioned is not recognised.
620-630	search through all 25 objects to see if it is in the current location. If E% is zero, the object is not there and control is returned to the main program loop.
640	set the value of A(R) to 1 for object number R.
650	check if the object is the 'hyper viper' and unless the player is wearing the gloves (A(11)=2), return to the main loop.
660	check to see if the object cannot be picked up, print a message and return to the main loop.
670	check to see if the object cannot be picked up, print a different message and return to the main program.
680	set the flag E% to zero and search all four elements of the array V$(X) to find an empty space.
690	if an empty element is found, store the description of the object in it and set the value of X to 5 so as to terminate the loop. Also set E% to one.
700	if E% is still zero, then the array V$(X) is full and the player can't carry any more objects until he drops one.
710	set the pointer B%(N%(R)) for the object to zero, so that it disappears from view and return to main loop.

In this game, the player is allowed to carry only four items at any one time. As soon as the array V$(X) is full, the player will be unable to carry any more objects. If you want to change this to allow five items to be carried, you will need to change line 680 to

680 E%=0:FOR X=1 TO 5

You will, in addition need to make similar changes in the 'drop' and 'inventory' routines.

Split the input sentence and check items

```
720 REM ** split sentence and check items **
730 L$="":XX=INSTR(Z$," "):R=0
740 L%=0:L$=RIGHT$(Z$,(LEN(Z$)-XX))
750 IF LEN(L$)<2 THEN RETURN
760 FOR X=1 TO 25:IF LEFT$(N$(X),LEN(L$))=L$ THEN
L%=1:R=X
770 NEXT:RETURN
```

This routine is exactly the same as that used in the other games, except for the number of objects checked for in line 760. In this game, there are 25 objects and the program must search through all of them to find a match between the object's description and the word typed in by the player. For more explanation, see the description of the same routine in *The Wizard's Quest*. Do remember, however, that the line numbers will be different!

INVENTORY

```
780 REM ** inventory **
790 E=0:PRINT"You are carrying :-"
800 FOR X=1 TO 4:IF V$(X)<>"" THEN PRINT V$(X):E=1
810 NEXT:IF E=0 THEN PRINT"Not a sausage!"
820 IF A(4)=2 THEN PRINT"You are wearing the space
 suit!"
830 IF A(11)=2 THEN PRINT"You are wearing the leat
her gloves!"
840 PRINT
850 RETURN
```

Line

790	set flag to zero and print message.
800	search all four elements of V$(X) and if they are not empty, print description of object carried and set the flag to 1.
810	if flag is still zero, print message 'not a sausage!'.
820	check to see if wearing the space suit.

830 check to see if wearing the leather gloves.
840 print blank line to leave space on screen.
850 return to the main program.

Few changes have been made to this routine. The message printed when the player is not carrying anything has been changed and the two tests to see whether the player is wearing anything are included. In this game, the player must be wearing the space suit before pressing the button on the door of the airlock, otherwise the poisonous gas will kill him. Once he has pressed this button, however, he will be able to take off the space suit. You may like to change this by inserting a line into the main control loop of the program such as

225 IF P%>3 AND A(4)<‹2 THEN E$="You breath the atmosphere and die in agony!!!":GOSUB 1330

A(4) would be set to zero again by dropping the space suit, whilst the value of A(11) would be set to zero by dropping the leather gloves. Thus dropping the space suit in any location greater than number 3 would result in death!

DROP

```
860 REM ** drop **
870 GOSUB 730:IF L%<1 THEN PRINT"You don't have a
";L$:RETURN
880 E%=0:FOR X=1 TO 4
890 IF V$(X)=G$(N%(R)) THEN V$(X)="":E%=1
900 NEXT:IF E%=0 THEN PRINT"You are not carrying i
t!":RETURN
910 B%(N%(R))=P%
920 A(R)=0
930 IF R=10 AND P%=39 THEN S%(39,2)=40:Q$(39)=LEFT
$(Q$(39),24):PRINT"The viper attacks the dog and d
rives it away!":B%(10)=0
940 IF R=9 AND P%=38 THEN PRINT"The guard goes nut
s over them and moves aside to let me in!":S%(38,3
)=39:Q$(38)=LEFT$(Q$(38),74):B%(9)=0
950 RETURN
```

Line

870 call the subroutine to split the input sentence and return the number of the object mentioned in R.
880 search all four items being carried.
890 if object carried is equal to the object mentioned, remove its description from V$(X) and set E=1.
900 if E is still zero, player is not carrying the object.
910 set pointer for the location of the object to P%.

920 set flag A(R) to zero so that the computer knows that the player is no longer carrying it.
930 check to see if the viper is dropped in location 39.
940 check to see if the nuts are dropped in location 38.
950 return to the main program control loop.

One difference which you will probably have noticed between this program and the other games in this book is that all the responses are written in the second person, rather than the first person. This is very much a matter of personal taste. In this program for example, you will be given messages such as:

'You are not carrying a lamp' or 'You can't pull a button!'

In the previous games, these messages would have been :

'I am not carrying a lamp' and 'I can't pull a button!'

The only other differences between this subroutine and the 'drop' routines in the other games lie in lines 930 and 940. The player must drop the viper in location 39 to drive the dog away. This is another example where the dog is mentioned only in the description of the location and where the description of the location is shortened when the dog has gone. You must bear this in mind when changing the data file.

In line 940, a check is made to see whether the player has dropped the nuts in location 38. The guard then goes 'nutty' and the map is changed to allow movement east. In addition, the description of the location is shortened and the pointer which tells the computer in which location the nuts are found is changed to zero.

WEAR

```
960 REM ** wear **
970 GOSUB 730:IF R=11 AND A(11)=1 THEN A(11)=2:PRI
NT"O.K.":RETURN
980 IF R=4 AND A(4)=1 THEN A(4)=2:PRINT"O.K.":RETU
RN
990 IF R=11 OR R=4 THEN PRINT"You haven't got it!"
:RETURN
1000 PRINT"You can't wear ";L$:RETURN
```

Line

970	call the subroutine to split the input sentence and return the value of R corresponding to the number of the object, check if object is gloves (number 11) and that they are carried, change the flag A(11).
980	check if the object is the space suit and that it is being carried, set the flag A(4) and return to main program.
990	if R=4 or R=11, print message and return to main program.
1000	print message and return to main loop.

In this game, the player must wear the gloves before being able to get the 'hyper viper' and must wear the space suit before pressing the button on the airlock. No other objects within the game can be worn. The program will reach line 1000 only if the player attempts to wear something stupid! This must be borne in mind when modifying the data file. If you do decide to change object 4 or object 11, then it *must* be changed into something which can be worn.

GO IN

```
1010 REM ** go in **
1020 IF P%=6 AND A(18)=0 THEN PRINT"You can't sque
eze past the boulder!":RETURN
1030 IF P%=6 THEN P%=9:PRINT"O.K.":RETURN
1040 IF P%=12 THEN P%=11:PRINT"O.K.":RETURN
1050 PRINT"You can't!":RETURN
```

There are just two places in this game where movement into a new location is allowed. Studying the map will show you that movement from location 6 takes you to location 9, whilst movement from location 12 takes you to location 11.

Line

1020	if in location 6 and the flag A(18) is still zero, the player can't get past the boulder and control is returned to the main loop.
1030	if in location 6, move to location 9 and return to the main loop.
1040	if in location 12, move to location 11 and return to the main loop.
1050	print the message about movement being impossible and return to the main loop.

The main puzzle in this section of the game is how to get past the boulder and into the cave in location 6. The boulder, object 18, cannot be moved and the solution lies in blasting it with the phaser. Because the boulder is one of the objects which you can't pick up, the value of A(18) would not normally be 1. Instead of introducing yet another flag, I decided to use this and change its value to 1 when the phaser is fired at the boulder. If, therefore, the boulder has not been removed, A(18) will still be zero and line 1020 will prevent movement into the cave. If you do decide to change the data file, don't change the description of the boulder without modifying the message in line 1020.

If the player types 'in' or 'go in' and he is not in one of the two rooms where this is possible, the program will reach line 1050 and the message 'You can't' will be printed.

GO OUT

```
1060 REM ** go out **
1070 IF P%=9 THEN P%=6:PRINT"O.K.":RETURN
1080 IF P%=11 THEN P%=12:PRINT"O.K.":RETURN
1090 PRINT"You can't!":RETURN
```

Line

1070	if in location 9, move to location 6, print the message and return to the main loop.
1080	if in location 11, move to location 12, print the message and return to the main loop.
1090	print the message and return to the main loop.

This routine is complementary to the previous subroutine and works only in location 9 and 11. If the player is not in either of these places, line 1090 is reached and a message about his ability to go in is printed before control is returned to the main loop.

FIRE PHASER

```
1100 REM ** fire phaser **
1110 GOSUB 730:IF R<>2 THEN PRINT"You can't fire a
 ";L$:RETURN
1120 IF A(2)<>1 THEN PRINT"You haven't got it!":RE
TURN
1130 IF P%<>6 THEN PRINT"That would be too dangero
us here!":RETURN
1140 PRINT"That does the trick!":B%(18)=0:A(18)=2:
RETURN
```

As mention earlier, the way past the boulder is to blast it with the phaser and this is the routine which controls that action. It is called from the main loop whenever the player tries to 'blast', 'fire' or 'use' an object.

Line

1110	call the subroutine to split the sentence into two words. If the second word is not object 2, the phaser, a message is printed and control returned to the main loop.
1120	check to see if the player is carrying the phaser and return to the main loop if not.
1130	check the location and if it isn't number 6, return to the main loop.
1140	print the message, change the pointer to the location of object 8 so that it disappears, change the flag A(18) to 2 and return to the main loop.

The program firstly checks whether you are carrying the phaser and then whether the location is correct. Only if both conditions are all right does the program reach line 1140, where the flag A(18) is set to 2. Remember that the value of this flag is tested when you attempt to enter location 6 and therefore if you do change the data file, you would be well advised to change the boulder to something else which you need to shoot to get past (a soldier perhaps).

GO DOWN

```
1150 REM ** go down **
1160 IF P%=34 THEN P%=35:PRINT"O.K.":RETURN
1170 IF P%=5 THEN PRINT"The ground's too far below
 you!":RETURN
1180 PRINT"Don't be silly!":RETURN
```

Location 34 is the only place in this game where this instruction works.

Line

1160	check to see if the player is in location 34, move him to location 35 and return to the main loop.
1170	if in location 5, print the message and return to the main loop.
1180	print the message about the stupidity of trying to go down and return to the main loop.

From location 34, going down takes you to location 35. The player may well attempt to go down from location 5, but in this game he must jump !

GO UP

```
1190 REM ** go up **
1200 IF P%=35 THEN P%=34:PRINT"O.K.":RETURN
1210 IF P%=7 THEN P%=5:PRINT"O.K.":RETURN
1220 PRINT"not here!":RETURN
```

Line

1200	if in location 35, move to location 34 and return to the main loop.
1210	if in location 7, move to location 5 and return to the main loop.
1220	print the message that the action is not possible and return to the main loop.

Although there is only one location where the player can go down, I have allowed them to go up into the spaceship from location 7 to location 5, whilst they must jump to go the other way! (after all the gravity is low on this planet!). You may like to change this by adding an extra subroutine to enter the ship again.

JUMP

```
1230 REM ** jump **
1240 IF P%=5 THEN P%=7:PRINT"Phew safe landing! Th
e gravity must be  low!":RETURN
1250 IF P%=7 THEN P%=5:PRINT"The gravity is so low
, you made it!":RETURN
1260 PRINT"not here!":RETURN
```

Line

1240	if in location 5, move to location 7, print the message and return to the main loop.
1250	if in location 7, move to location 5, print the message and return to the main loop.
1260	print the message about the futility of jumping and return to the main loop.

There are, thus, two ways back into the spaceship. The player can either 'go up' or 'jump'. This is only made possible by the low gravitational forces on the planet. You will also notice that the message printed in line 1240 refers to the force of gravity. If you do intend to change the description of location 5 in the data file, you should ensure that it still includes a clue about jumping. You may also like to change the message printed in line 1240.

PRESS

```
1270 REM ** press **
1280 GOSUB 730:IF R<>5 THEN PRINT"What do you want
 me to press?":RETURN
1290 IF P%<>3 THEN PRINT"Not here!":RETURN
1300 IF A(4)<>2 THEN E$="Whoosh! The airlock opens
 and you die in the poisonous atmosphere!":GOSUB 1
340
1310 PRINT"The airlock opens!":S%(3,2)=4
1320 RETURN
```

Line

1280	call the subroutine to split the player's sentence into two words. The number of the object mentioned (R) is then checked and if it isn't 5, the button, a message is printed and control is returned to the main control loop.
1290	check the current location and if it isn't 3, print the message and return to the main loop.
1300	check the flag A(4) to see if the player is wearing the space suit and call the death routine if not.
1310	print the message and change the map.
1320	return to the main control loop.

The problem of how to get out of the spaceship has already been mentioned and you will need to take great care when changing the data file that object 5 remains something which must be pushed and that the player must be wearing object 4 first !

Notice that the description of the way in which death occurs is stored in the variable E$ before the death routine is called in line 1340.

LOSE GAME

```
1330 REM ** lose game **
1340 CLS:PRINT E$:LOCATE 1,10:PRINT"Press the <Spa
ce Bar> for another game."
1350 A$=INKEY$:IF A$<>" " THEN 1350
1360 RUN
```

Line

1340	clear the screen, print the description of death held in the variable E$ and print the message about pressing the space bar.
1350	wait for the space bar to be pressed.
1360	run the program from the start again.

One point worth noting about this game is that the player will need to reload the data file from the start if he loses the game because the contents of the arrays will have been changed during play.

PULL

```
1370 REM ** pull **
1380 GOSUB 730:IF R<>6 THEN PRINT"You can't pull a
 ";L$:RETURN
1390 IF P%<>4 THEN PRINT"not here!":RETURN
1400 E$="The ship explodes. You have just pulled t
he self destruct lever!":GOSUB 1340
```

This routine was written to lure the unwary player to his instant death!

Line

1380	call the subroutine to split the player's instructions into two words; if the second word is not the lever, print message and return to main program.
1390	if not in location 4, print message and return.
1400	set the contents of E$ to hold message and call the death subroutine.

There is only one object in this game which can be pulled, the lever, and checks are made that the player has mentioned it and that they are in the correct location. Do remember to change the lever into some other object which needs to be pulled if you attempt to change the data file. You may also like to change the message to better describe the method of death!

CLIMB

```
1410 REM ** climb **
1420 IF P%=20 OR P%=23 THEN 1430 ELSE PRINT"Not he
re!":RETURN
1430 IF A(17)=0 THEN E$="You slip from the rope an
d fall to your death. If only you had used a "+G$(
17):GOSUB 1340
1440 IF P%=20 THEN P%=23:PRINT"O.K.":RETURN
1450 IF P%=23 THEN P%=20:PRINT"O.K.":RETURN
```

There is a rope stretching between locations 20 and 23. Climbing is not possible in any other locations in this game. Players who try climbing across the rope without holding the large hook, object 17, will slip from the rope and fall to their death.

Line

1420	check location and if climbing not possible, print the message and return to the main loop.
1430	check the value of the flag A(17) to see if the hook is being carried, set the message string and call the death subroutine if necessary.
1440	if in location 20, move to location 23, print message and return to the main program loop.
1450	if in location 23, move to location 20, print message and return to the main program loop.

I have included a clue in line 1430, to help the player overcome the problem of crossing the rope next time. You may like to try experimenting with sound effects when the player falls from the rope in line 1430.

CUT

```
1460 REM ** cut **
1470 IF P%<>19 THEN PRINT"That's not the right app
roach!":RETURN
1480 IF A(1)=0 THEN PRINT"You need a knife!":RETUR
N
1490 S%(19,2)=20:Q$(19)=LEFT$(Q$(19),28):RETURN
```

Once the player has reached location 19 in this game, he will be unable to progress further through the jungle without cutting his way through the dense undergrowth. He must, of course, be carrying a knife in order to do this.

Line

1470	check current location and print message / return to main loop if not in location 19.
1480	check to see if carrying the knife (object 1) and print message / return to main loop if not.
1490	change map, shorten the description of the location and return to the main program.

Don't forget that if you change the data file, the knife must be changed into something to cut with, a saw or an axe perhaps. In addition, the description of location 19 should contain the clue that cutting a way through will be necessary!

SCORE

```
1500 REM ** check score **
1510 S%=0:IF SC=1 THEN S%=S%+2
1520 IF SD=1 THEN S%=S%+1
1530 IF SE=1 THEN S%=S%+1
1540 IF SF=1 THEN S%=S%+1
1550 IF SG=1 THEN S%=S%+1
1560 RETURN
```

Unlike the other subroutines described in this chapter, this one is called *every* time the program goes round the main control loop, so that the computer always has an up to date record of the player's score.

Line

1510	set score to zero, if flag SC=1 then increase the score by 2.
1520	if the flag SD=1 then increase the score by 2.
1530	if the flag SE=1 then increase the score by 2.
1540	if the flag SF=1 then increase the score by 2.
1550	if the flag SG=1 then increase the score by 2.
1560	return to the main program control loop.

Scoring in this game is achieved in a totally different way from the previous games. The value of S% is increased by 2, so increasing the score by 20%, for each of the five problems solved. These five problems are all associated with the spaceship and are described later. Each one solved sets the value of a flag (SC to SG) to 1.

INSERT

```
1570 REM ** insert **
1580 GOSUB 730:IF R=8 OR R=25 THEN GOTO 1600
1590 PRINT"Don't be ridiculous!":RETURN
1600 IF A(8)=1 AND P%=2 THEN PRINT"You insert the
crystal into its socket!":SC=1:GOSUB 1640:RETURN
1610 IF A(25)=1 AND P%=50 THEN GOSUB 1670:RETURN
1620 PRINT"You can't do that yet!":RETURN
1630 REM ** get rid of the crystal **
1640 FOR X=1 TO 4:IF V$(X)=G$(8) THEN V$(X)=""
1650 NEXT:B%(12)=0:RETURN
1660 REM ** insert credit card **
1670 IF A(7)=0 THEN PRINT"It pours all over the fl
oor!":RETURN
1680 SD=1:PRINT"You fill it up with fuel!":RETURN
```

There are two objects which need to be 'inserted' in this game, namely the intergalactic credit card, object 25, and the crystal warp control, object 8. The routine is fairly complex, which makes it a little more difficult to modify these objects in the data file, whilst still keeping a sense of logic in the game.

Line

1580-1590	call the subroutine to split the player's sentence into two words. If the second word is not the crystal or the credit card, print message and return.
1600	check whether player is in location 2 and carrying the crystal. If he is, print message, set flag SC for score, call the subroutine to drop crystal and return to main program.
1610	if player wants to drop the credit card in the correct location, call subroutine to do it and return to the main program control loop.
1620	print the message that the action is not yet possible and return to the main control loop.
1640	search through the four items being carried (V$(X)) and remove the crystal.
1650	set the pointer for the empty socket to zero so that it disappears from view.
1670	if you are not carrying the can, item 7, the fuel pours all over the floor and control is returned to the main loop.
1680	set the flag SD to one, print the message that the can is full of fuel and return to the main loop.

Should you decide to modify the data file, you will need to make sure that the can, item 7, is changed for something which needs filling with a liquid and that you need to 'insert' item number 25 into a slot before that liquid is dispensed. The message printed in line

1680 was deliberately kept short so as to be applicable even if the data file were changed. You may like to change it to a more detailed description.

The score in this routine will be increased by 2 when the player inserts the crystal warp control into the empty socket found in location 2, the cabin of the spaceship. The score is also increased by 2 when the player inserts the credit card into the slot found in location 50 and collects the rocket fuel in the can.

FILL

```
1690 REM ** fill **
1700 GOSUB 730:IF R=7 AND P%=50 THEN PRINT"nothing
 comes out!":RETURN
1710 IF R=7 THEN PRINT"Not here!":RETURN
1720 IF P%=2 AND A(7)=1 AND SD=1 THEN PRINT"You fi
ll the fuel tanks!":SE=1:RETURN
1730 PRINT"You can't do that just yet!":RETURN
```

Before being able to escape from the planet, the ship has to be repaired and filled with fuel. To do this, the player must be carrying the can full of fuel.

Line

1700	call subroutine to split the input sentence into two words. If the second word refers to the can and the player is in location 50, nothing comes out and control passes back to the main program loop.
1710	if the player tries to fill the can, object 7, print the message and return to the main loop.
1720	if player is in location 2 and carrying the can and the can is full (SD=1), print message, set the value of the flag SE to increase the score and return to the main program.
1730	print message and return to the main loop.

The first part of this subroutine checks whether the player is attempting to fill the can from the fuel tank. As we have already seen, the way to do this is to insert the credit card into the slot and this means that we must prevent the player from filling the can in location 50. Line 1720 increases the score by 2, which is equivalent to a score of 20%, if the player is in location 2 and carrying a full can of fuel.

SAVE GAME

```
1740 REM ** save game **
1750 CLS:PRINT"Please insert a tape and set ready
to   record!"
1760 PRINT:PRINT"Press <Space Bar> when ready"
```

```
1770 A$=INKEY$:IF A$<>" " THEN 1770
1780 OPENOUT"data"
1790 FOR x=1 TO 50:PRINT#9,Q$(x):NEXT
1800 FOR x=1 TO 50:FOR y=1 TO 4:PRINT#9,S%(X,Y):NE
XT Y,X
1810 FOR x=1 TO 25:PRINT#9,G$(X):NEXT X
1820 FOR x=1 TO 25:PRINT#9,B%(X):NEXT X
1830 FOR x=1 TO 25:PRINT#9,N$(X):NEXT X
1840 FOR x=1 TO 25:PRINT#9,N%(X):NEXT X
1850 FOR x=1 TO 25:PRINT#9,A(X):NEXT X
1860 FOR x=1 TO 4:PRINT#9,V$(X):NEXT X
1870 PRINT#9,SA,SB,SC,SD,SE,SF,SG,SH,P%
1880 CLOSEOUT:RETURN
```

This routine is identical to the routine in the data file creating program. It is used to write a full data file containing the player's new position and all the other variables used in the game.

Line

1750-1760	print message to insert the data tape and wait for the space bar to be pressed.
1770	wait for the space bar to be pressed.
1780	open the file with a file name 'data' for saving the data.
1790	save the descriptions of the 50 locations in the game.
1800	save the current map.
1810	save the current descriptions of the 25 objects.
1820	save the 25 pointers to the current location of the objects found in the game.
1830	save the 25 words understood on tape.
1840	save the pointers to the words understood.
1850	save the 25 flags of the objects being carried.
1860	save the descriptions of the four objects being carried.
1870	save the flags SA to SH and the current position P%.
1880	close the file and return to the main program loop.

It is important to note that the descriptions of some of the locations and objects will change during the play of the game and therefore it makes sense to save the data for all the locations, objects and flags found in the game.

LOAD GAME

```
1890 REM ** load game **
1900 CLS:PRINT"Please insert the data tape into th
e    recorder."
1910 OPENIN"data"
1920 FOR x=1 TO 50:INPUT#9,Q$(x):NEXT
1930 FOR x=1 TO 50:FOR y=1 TO 4:INPUT#9,S%(X,Y):NE
XT Y,X
```

```
1940 FOR x=1 TO 25:INPUT#9,G$(X):NEXT X
1950 FOR x=1 TO 25:INPUT#9,B%(X):NEXT X
1960 FOR x=1 TO 25:INPUT#9,N$(X):NEXT X
1970 FOR x=1 TO 25:INPUT#9,N%(X):NEXT X
1980 FOR x=1 TO 25:INPUT#9,A(X):NEXT X
1990 FOR x=1 TO 4:INPUT#9,V$(X):NEXT X
2000 INPUT#9,SA,SB,SC,SD,SE,SF,SG,SH,P%
2010 CLOSEIN:RETURN
```

This subroutine is called right at the start of the game to load in the data file containing the starting position. It can also be used to load a game which has been saved during the course of play.

Line

1900	clear the screen and print message to insert the data tape into the recorder.
1910	open the channel to input the data file.
1920	load in the description of the 25 locations.
1930	load in the array used to hold the map.
1940	load in the descriptions of the 25 objects.
1950	load in the pointers for the locations where the objects are to be found.
1960	load in the words recognised.
1970	load in the pointers to the words recognised.
1980	load in the flags for the objects carried.
1990	load in the descriptions of the four objects carried.
2000	load in the flags SA to SH and the current location P%.
2010	close the file and return to the main program.

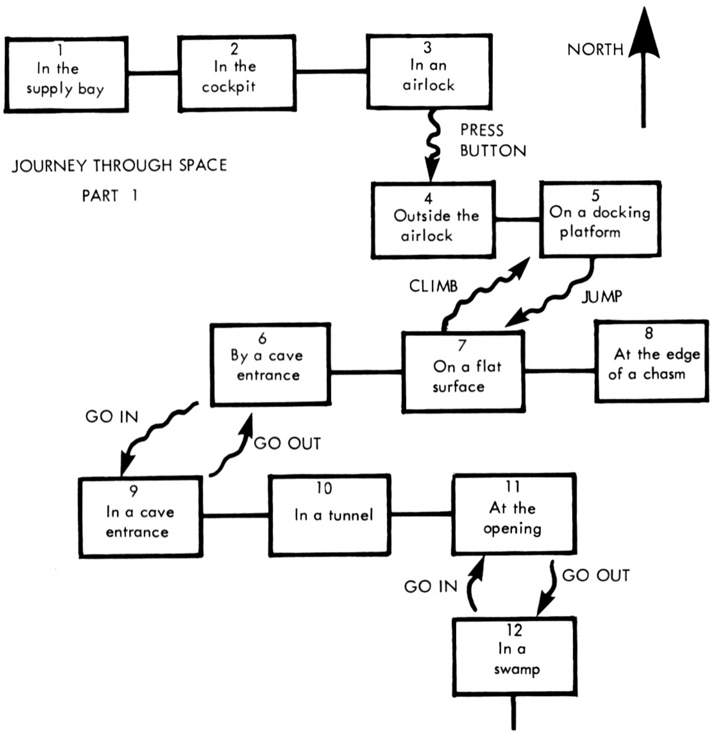
If you compare this routine with the SAVE GAME routine, you will see that the data is read in from tape or disc in *exactly* the same order. Any error in this section, however slight, will prevent the game working at all and you must check that there are no typing errors when entering it into your computer. You should now have typed all sections of the main game into your computer and must check carefully for any typing errors before saving a copy onto tape or disc. You will not be able to test this game out by running it until you have a data file on tape. In the next chapter you will find the listing for the data file creating program, and this must be typed in and RUN. You will be asked to insert a tape into the recorder and you would be advised to save the data file created by the program immediately after the main game on the first tape.

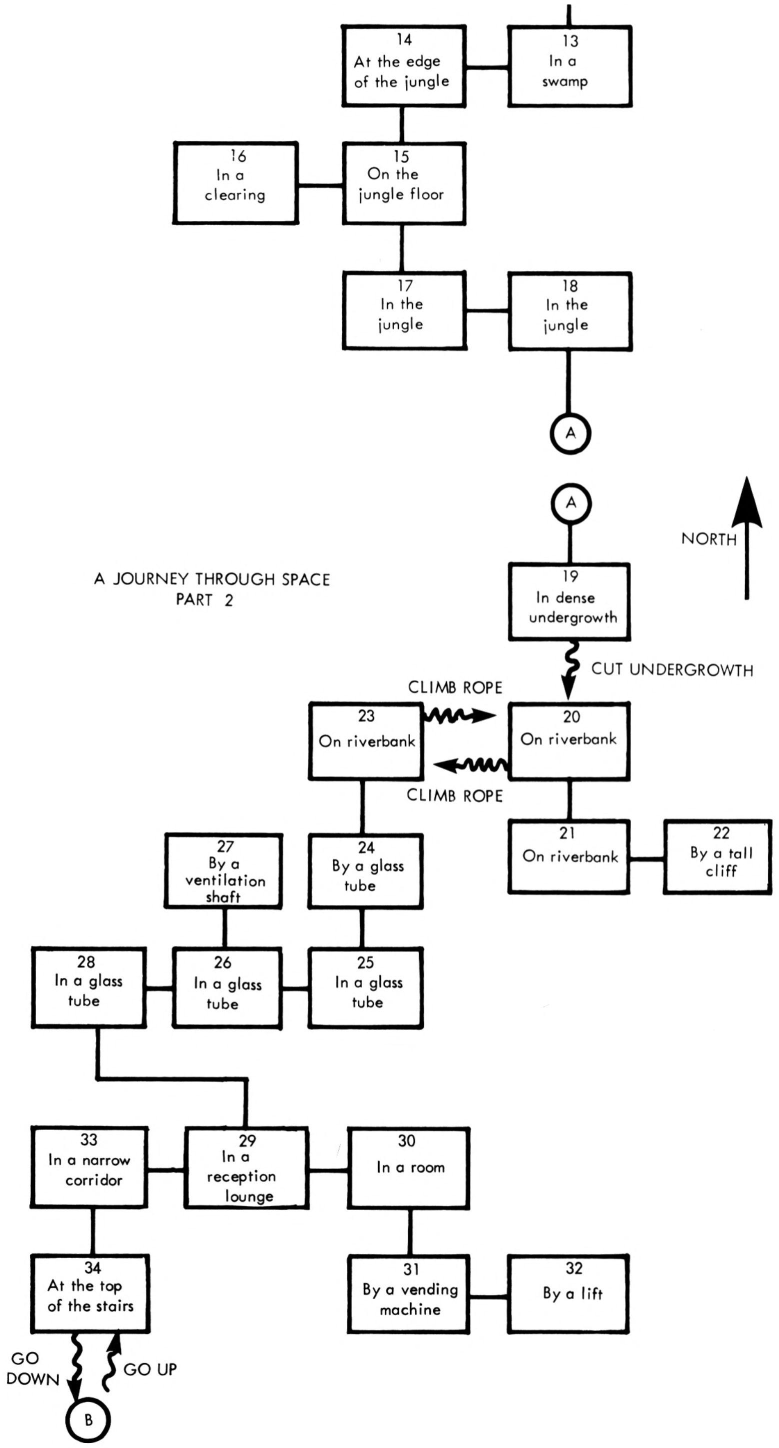
Because you will not be able to test each section of the program as it is typed in, you must take extra care with data entry and check each section against the printed listing before going on to the next section.

Creating the data file 15

Before looking at how the data file is created, we need to draw the map of the game in the same way as with the previous two games. Although the program listing here allows you to modify the descriptions of the locations, it doesn't, as it stands, allow you to type in changes to this map. Only a minor modification to the program would be necessary to allow this to be done and I will explain in further detail how to set about it.

The map





14 At the edge of the jungle
13 In a swamp
16 In a clearing
15 On the jungle floor
17 In the jungle
18 In the jungle
A
A
NORTH
A JOURNEY THROUGH SPACE PART 2
19 In dense undergrowth
CUT UNDERGROWTH
CLIMB ROPE
23 On riverbank
20 On riverbank
CLIMB ROPE
27 By a ventilation shaft
24 By a glass tube
21 On riverbank
22 By a tall cliff
28 In a glass tube
26 In a glass tube
25 In a glass tube
33 In a narrow corridor
29 In a reception lounge
30 In a room
34 At the top of the stairs
31 By a vending machine
32 By a lift
GO DOWN
GO UP
B

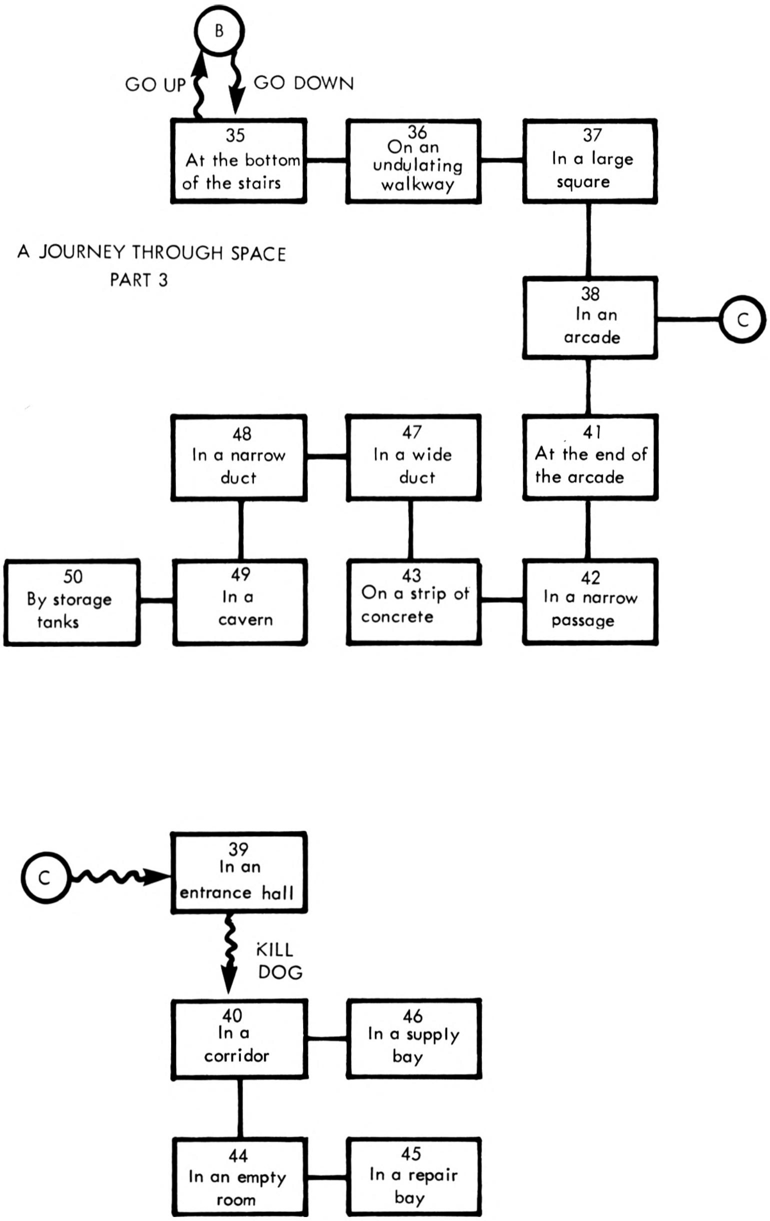


Fig. 15.1 Map for *A Journey through Space*.

Careful study of this map will show you that there are 50 locations. The data for the descriptions of the locations and the objects must first be read into the arrays. I have again used the same variable names for these arrays, so as to avoid confusion when trying to debug the program.

Reading the data

```
10 REM ** data file creating program **
20 CLS:LOCATE 10,2:PRINT"Data File Creator"
30 REM ** must be used in conjunction with the adv
enture program  **
40 REM ** READ the DATA for 'A Journey through Spa
ce' **
50 DIM Q$(50),S%(50,4),V$(4),G$(25),B%(25),N%(25),
N$(25),A(25)
60 FOR X=1 TO 50:READ Q$(X)
70 FOR Y=1 TO 4:READ S%(X,Y): NEXT Y,X
80 DATA in the supply bay.,0,0,2,0
90 DATA in the cockpit of the spaceship.,0,0,3,1
100 DATA in a small airlock.,0,0,0,2
110 DATA outside the airlock.,3,0,5,0
120 DATA on a docking platform high above the    s
urface of the planet.,0,0,0,4
130 DATA outside a cave entrance.,0,0,7,0
140 DATA on the flat surface of the planet Lucia.T
he spaceship is here.,0,0,8,6
150 DATA at the edge of a deep chasm. Travel to  t
he east is not possible.,0,0,0,7
160 DATA inside a gloomy cavern. A dark tunnel   l
eads east.,0,0,10,0
170 DATA in a dark tunnel. Drips of water keep   f
alling on my head.,0,0,11,9
180 DATA at the cave entrance. I can see a fetid s
wamp in the distance.,0,0,0,10
190 DATA on a narrow footpath leading through thef
etid swamp. Swirls of purple mist rise from the sw
amp. A cave can be seen here.,0,
13,0,0
200 DATA at the edge of a fetid swamp. A dry     f
ootpath leads north through the purple mist.,12,0,
0,14
210 DATA at the edge of a thick jungle.,0,15,13,0
220 DATA on the jungle floor.Strange insects     c
rawl over my feet.,14,17,0,16
230 DATA in a clearing. Thick undergrowth stops  m
e going further west.,0,0,15,0
240 DATA on a muddy trail leading through a densej
ungle.,15,0,18,0
250 DATA on a muddy path. The trees are alive    w
ith strange creatures.,0,19,0,17
260 DATA at the end of a narrow path. It looks asi
f nobody has travelled this way for a  long time b
ecause the undergrowth is so den
se to the south.,18,0,0,0
270 DATA on the banks of a narrow river. A rope  s
tretches across to the far side.,19,21,0,0
280 DATA on the banks of a fast flowing river. A h
igh cliff towers above me.,20,0,22,0
290 DATA underneath a tall cliff. A cave entrancec
an be seen above my head.,0,0,0,21
```

```
300 DATA on the banks of a river of mercury. A   r
ope stretches across to the far side.,0,24,0,0
310 DATA on a narrow path leading into a large   g
lass tube.,23,25,0,0
320 DATA in a clear glass tube.,24,0,0,26
330 DATA in a wide glass tube high above the     p
lanet surface. A door leads north.,27,0,25,28
340 DATA in a ventilation shaft. It is too narrowt
o go further north,0,26,0,0
350 DATA in a wide glass tube leading into the   t
op of a large building.,0,29,26,0
360 DATA in the reception lounge of the 'Lucia   M
ining Corporation' headquarters.,28,0,30,33
370 DATA in a small room full of chairs covered  w
ith a glowing purple fabric.,0,31,0,29
380 DATA by a vending machine. The two slots are c
overed by a red notice written in a    strange lan
guage,30,0,32,0
390 DATA by a lift. The doors are closed and the l
ights are not working. I can't see any switches or
 buttons to press.,0,0,0,31
400 DATA in a narrow corridor. The walls are     l
ined with wierd plants with eyes which follow my e
very move.,0,34,29,0
410 DATA at the top of a flight of stairs. A     p
lastic android stands at the top.,33,0,0,0
420 DATA at the bottom of a flight of stairs. A  w
ide passage leads east.,0,0,36,0
430 DATA on a slowly undulating walkway.,0,0,37,35
440 DATA in a large square.,0,38,0,36
450 DATA in a golden arcade. Small blue trees line
 the sides of an enormous statue. A nutty guard wo
n't let me into the green    bui
lding.,37,41,0,0
460 DATA inside an entrance hall. A mad Lucian   R
ock Hound spits molten gold at me.,0,0,0,38
470 DATA in a narrow corridor. Doors lead south  a
nd east.,39,44,46,0
480 DATA at the end of the arcade. A path leads  s
outh between two buildings.,38,42,0,0
490 DATA in a narrow passage between tall        b
uildings.,41,0,0,43
500 DATA on a narrow strip of concrete at the    e
dge of a sheer drop.,47,0,42,0
510 DATA in an empty room. A door leads east.,40,0
,45,0
520 DATA in a repair bay. It's full of tools.,0,0,
0,44
530 DATA in a supply bay. A robot stands at the  c
ounter and looks at me.,0,0,0,40
540 DATA in a wide duct.,0,43,0,48
550 DATA in a narrow duct.,0,49,47,0
560 DATA in a large cavern full of storage tanks.,
48,0,0,50
```

```
570 DATA in the fuel storage bay. A large fuel   d
ispenser with a slot in it stands here.,0,0,49,0
580 FOR x=1 TO 25:READ G$(X),B%(X),N$(X):N%(X)=X:N
EXT X
590 DATA a strong knife,1,knife,a phaser,1,phaser,
a shovel,1,shovel,a space suit,1,suit
600 DATA a button,3,button,a lever,4,lever,a large
 can,22,can,a crystal warp control,46,crystal
610 DATA a packet of wolf nuts,32,nuts,a hyper vip
er,17,viper,a pair of leather gloves,2,gloves,a cr
ystal control socket,2,socket
620 DATA a fuel injection cap,2,cap,a damaged pane
l,2,panel,a repair manual,45,manual
630 DATA a remote control for androids,32,control,
a large hook,6,hook
640 DATA a boulder,6,boulder,a glowing statue,37,s
tatue,a ldoria plant,18,plant,an alien mask,24,mas
k,a metal bar,16,bar
650 DATA a fuel spout,50,spout,a slot,50,slot,an i
ntergalactic credit card,33,credit card
660 CLS:S%=0:P%=2
670 J$="A Journey through Space"
```

Line

20	clears the screen and prints the title of the program.
50	DIMension the arrays.
60	read in the descriptions of the locations into Q$(X).
70	read in the map into the array S%(X,Y).
80-570	DATA for the locations.

In a similar manner to the previous programs, each DATA line from line 80 to 570 contains a description of the location, followed by the numbers corresponding to the locations reached by going north, south, east and west respectively.

Line

580	read the description of the 25 objects, the number of the location in which they are found, the word by which they are recognised and set the pointer to the word.
590-650	DATA for the description, location and word recognised of the 25 objects.
660	clear the screen, set the score to zero and the location to number 2.
670	define the title of the game.

Reminder of the variables used

S%	holds the score
P%	holds the number of the current location
Q$(X)	holds the description of the location
S%(X,Y)	holds the map
G$(X)	holds the description of the object
B%(X)	holds the number of the location where the object is found
N$(X)	holds the word recognised by the computer as being connected with the object
N%(X)	pointer to the object

It is worth noting that the routine in the main program to read the data in does not contain a line to read in J$, the title of the game. If you do add a line to do this, a similar line must be added to the save game routine in the main program and to the file writing section in this program.

Change the data file?

```
680 REM ** change data or leave alone **
690 CLS:LOCATE 1,2:PRINT"Do you want to change the
 data file      <Y>es / <N>o ?"
700 AA$=INKEY$:AA$=LOWER$(AA$):IF AA$="n" THEN GOS
UB 730:PRINT"Data file saved now!":END
710 IF AA$="y" THEN GOSUB 900:GOSUB 730:PRINT"Data
 file saved now!":END
720 GOTO 700
```

When the program is run, the data for the standard game is read into the arrays and the section of coding between line 680 and line 720 asks the player whether they want to save the standard game, *A Journey through Space,* or whether they want to write their own data file.

Line

690	asks the question whether the player wants to change the data file.
700	wait for key to be pressed, if the player answers no, the subroutine to save the data is called and the program then ends.
710	if the player wants to change the data file, call the subroutine to change the data, save the data file and end the program.
720	jump back to test for a key being pressed.

Should the player decide not to change the data file, the information saved on the tape will contain all the data for *A Journey through Space*. Pressing the 'Y' key, however will take the player to the section of the program which allows him to type in changes to the descriptions of the locations, objects and words recognised.

SAVE GAME

```
730 REM ** save game **
740 CLS:PRINT"Please insert a tape and make ready
to  record."
750 PRINT:PRINT"Press <Space Bar> when ready."
760 A$=INKEY$:IF A$<>" " THEN 760
770 OPENOUT"data"
780 REM ** this line may be changed later ... see
notes **
790 FOR X=1 TO 50:PRINT#9,Q$(X):NEXT
800 FOR X=1 TO 50:FOR Y=1 TO 4:PRINT#9,S%(X,Y):NEX
T Y,X
810 FOR X=1 TO 25:PRINT#9,G$(X):NEXT
820 FOR X=1 TO 25:PRINT#9,B%(X):NEXT
830 FOR X=1 TO 25:PRINT#9,N$(X):NEXT
840 FOR X=1 TO 25:PRINT#9,N%(X):NEXT
850 FOR X=1 TO 25:PRINT#9,A(X):NEXT
860 FOR X=1 TO 4:PRINT#9,V$(X):NEXT
870 PRINT#9,SA,SB,SC,SD,SE,SF,SG,SH,P%
880 CLOSEOUT:RETURN
```

This section of the program *must* be absolutely identical to the section of code called in the main game when the player chooses to save a game during play. Any differences between these two routines will cause the DATA to be read in from tape in the wrong order and this will result in a game which doesn't make any sense, if it runs at all!

Line

740	clear the screen and print message to insert a tape ready to save the game.
750	print message to press the space bar when ready.
760	wait for the space bar to be pressed before saving the data onto tape.
770	open the cassette filing system.
780	see notes.
790	save the descriptions of the 50 locations.
800	save the map.
810	save the descriptions of the 25 objects.
820	save the pointer to the locations of the 25 objects.
830	save the words recognised for the objects.
840	save the pointers to the words recognised.

850 save the flags for the objects carried.
860 save the value of the flags SA to SH and the current location.
880 close the file and return.

Locomotive BASIC, as found on the Amstrad computers does not require the values of variables to be defined before they are used. In this case, the value of the flags A(X) and SA to SH will all be zero, although their values will change in the file saved when the player types SAVE during play.

When the data file is loaded from tape at the start of the game, the contents of J$ does, in fact, hold the title of the game. In all the routines for tape handling in Chapters 13 to 15, however, the value of J$ is not saved or loaded. It is fairly simple to add one extra line to each routine to do this. In this program, you will need to change the REM statement in line 780 to

```
780 PRINT #9,J$
```

and insert the following lines into the main program:

```
1785 PRINT #9,J$
1915 INPUT #9,J$
```

This will save the contents of J$ before any other data on the tape.

Changing the data

```
890 REM ** change data for a new game **
900 CLS:FOR X=1 TO 50
910 CLS:LOCATE 1,1:PRINT"Location number ";X
920 LOCATE 1,3:PRINT"Old description "
930 LOCATE 1,4:PRINT Q$(X)
940 LOCATE 1,10:PRINT"What is the new description
"
950 INPUT Q$(X)
960 PRINT"Is this correct <Y>es / <N>o ?"
970 AA$=INKEY$:AA$=LOWER$(AA$):IF AA$="n" THEN 910
980 IF AA$="y" THEN 990 ELSE 970
990 NEXT X
1000 FOR X=1 TO 25
1010 CLS:LOCATE 1,1:PRINT"Object number ";X
1020 LOCATE 1,3:PRINT"Old description :-"
1030 LOCATE 1,4:PRINT G$(X)
1040 LOCATE 1,10:PRINT"What is the new description
 "
1050 INPUT G$(X)
1060 PRINT:PRINT"What word will it be recognised b
y      ";:INPUT N$(X)
1070 PRINT"Is this correct <Y>es / <N>o ?"
1080 AA$=INKEY$:AA$=LOWER$(AA$):IF AA$="n" THEN 10
10
```

```
1090 IF aa$="y" THEN 1100 ELSE 1080
1100 NEXT x
1110 RETURN
```

When this section of program is reached, you will be shown a description of all 50 locations and will be asked to type in a new description. You should try to make sure that no words are split across two lines on the screen, as this will make for an untidy display when the game is run. Once you have typed in a new description and pressed ‹ENTER›, you will be asked whether this description is correct and if you press the 'N' key, you will be asked to type the description in again.

Do try to bear in mind the puzzles set in the game when making the changes to the descriptions. It would be a very stupid game indeed if the player had to jump from a flat piece of earth or if they had to climb across a footpath!

Once you have typed in the new descriptions of the locations, you will then be shown the current descriptions of the 25 objects found in the game and will be asked to type in their new description, together with the word by which they are recognised.

It is again important to make sure that, when you change descriptions of objects within the game, they are changed to something which makes sense within the context of the game. Do bear in mind that certain objects are associated with a particular command. The knife, for example, must be used to cut your way through the dense undergrowth and if you were to change it to a tortoise, how could you find your way through the dense growth?

Line

900	clear the screen, repeat the loop 50 times.
910	clear screen and print the number of the location.
920	print message about the old description.
930	print old description of the locations.
940	print message to ask for input of the new description.
950	input the new description of the location.
960	print message to ask if this is correct.
970	get keyboard input and if player presses 'N' key, return to input the description again.
980	if player doesn't press the 'Y' key, jump back to keyboard input.
990	next description.
1000	repeat the loop for the 25 objects.
1010	clear the screen and print the number of the object.
1020	print the old description of the object.
1030	print the description of the object.
1040	print message to ask about the new description.
1050	input the new description of the object.

1060	print message about the word by which it is recognised and input the word recognised.
1070	print message to ask if it is correct.
1080	wait for key to be pressed, if player presses 'N' key, return to input words again.
1090	if key pressed is not 'Y', jump back to test the key being pressed.
1100	next object.
1110	return to the main program.

Do make sure that you take care when typing in the description of the objects that the words recognised by the objects are not duplicated. To illustrate this, consider a game where object number 7 is a red button and object number 11 is a blue button. Thus the contents of G$(X) would be :-

Object	*Description*	*Word*
7	A red button	Button
11	A blue button	Button

If you were to use the same word to recognise two different objects, the computer would search through and find a match *only* for the first occasion. If you do have two objects in your game which are very similar, such as those above, you *must* make sure that the two words recognised are different. In the above case, I would suggest that you use 'red button' and 'blue button' for N$(7) and N$(11) respectively.

Conclusions

There are a number of advantages of writing a program in which the data for the game is loaded in from tape or disc. These are summarised below.

1 The program will occupy less memory space, which means that it is possible to include more locations, objects, puzzles and problems. You can also include far more detail in the descriptions of the objects and locations because of the freedom given by extra free memory; in addition, you may well find enough space to add graphics as well.
2 The player will find it more difficult to cheat and solve the adventure by escaping from the game and listing it.

3 The program can be better structured, without making the game so easy to solve that the player wants to cheat. It always makes it easier to develop a program if the structure is sorted out properly at the start rather than allowing the program to grow at random during development of the game. However a typical 'spaghetti' style program will be much more difficult for the player to solve by listing it. At the same time, however, there is one major disadvantage of developing a program in which the data for the game has to be loaded in from tape every time — time. Each time you make a simple typing error and the program crashes, you will have to correct the mistake and then load in the tape again. This will take several minutes at the very minimum and unless you have a great deal of patience, the whole process can be very daunting. However, for those of you fortunate to own a disc drive, this process will take only a few seconds and the difference in time between loading the data from disc and READing it from DATA lines within the program will be very small. With Locomotive BASIC, there are no changes needed within the program, as the OPENIN, CLOSE IN, PRINT #9 and INPUT #9 commands will assume that a disc is fitted if you have the disc interface attached. You may, however like to change the messages about inserting a tape so that they specifically mention discs only.

Adding the final touches 16

So far, we have taken a close look at many of the standard features of adventure games, but have not spent much time examining those refinements which can truly transform a game into a masterpiece.

Function keys

The function of the keys on the numeric keypad can be redefined on the Amstrad computer and so far, we have completely ignored this facility. I recently played an adventure game where these keys were given definitions useful to the player. This game loads into the computer in two sections. The first program is a very short one and serves two purposes. Firstly, it prints the titles and secondly, it changes the definitions normally associated with the function keys into words which are of far more use to the adventurer. There is, in fact, a second advantage to be gained from this approach. We have, until now, been unable to include instructions within the game because the memory used up by them can be better utilised in setting puzzles and problems. If, however, the titles and instructions are included in a short program to define the function keys, we need have no such worries.The listing below shows how this could be done for *The Wizard's Quest.*

```
10 KEY 136,"go north"+CHR$(13)
20 KEY 134,"go east"+CHR$(13)
30 KEY 130,"go south"+CHR$(13)
40 KEY 132,"go west"+CHR$(13)
50 KEY 133,"inventory"+CHR$(13)
60 KEY 135,"help"+CHR$(13)
70 KEY 137,"score"+CHR$(13)
80 KEY 128,"search"+CHR$(13)
90 KEY 131,"pray"+CHR$(13)
100 KEY 128,"wait"+CHR$(13)
110 MODE 1
120 LOCATE 10,2:PEN 2:PRINT "The Wizard's Quest"
130 CHAIN"wizard"
```

Line	
10-100	define the function keys 1 to 10
110	set the screen mode
120	print the titles
130	load the main program

You will notice that I have defined the funcion keys to print some of the most common instructions used in adventure games. The PRINT CHR$(13) command at the end of each of these lines is used to enter the command into the computer whenever that function key has been pressed. Although the 'get' command is one of the most commonly used instructions in adventure games, this has not been assigned to a function key because the player would still have to type in the name of the object to be picked up. For that reason, the definitions given to the function keys are single word commands only.

The final line of this sort listing is used to load the main game into the computer. You must make sure, of course, that the main game has been saved with the same filename as that listed in line 130 of this program. If you do want to insert the instructions in this program, this should be done immediately before the command to load the main game.

Full sentence decoding

In all the games in this book, I have stuck to the traditional one or two word sentence recognition. Many commercial games now include the ability to understand far more complex sentences. Unfortunately, BASIC doesn't leave much room in memory for the inclusion of very complex sentence analysis, but you should be able to make a few improvements. The simplest of these would be to allow the inclusion of the word 'the' in the sentences so that the player can type 'get the rope'. This makes the game seem a little more realistic and should help to involve the player more in the game. In order to do this, a short section of code will need to be inserted into the routine used to split the sentence into two words. It is at this point that you really start to appreciate the inclusion of INSTR within the BASIC language. Adding this feature should prove to be an interesting, and not too difficult, exercise.

Data compression

Many commercial adventure games contain such detailed descriptions of objects and locations that it would be impossible to achieve in BASIC. Even storing the data for the game directly in memory would not allow the quality of description achieved by some programmers. Level 9 Computing's specially created adventure language A-CODE illustrates just what can be achieved in 32K of RAM. Just how this works, they haven't revealed, although we can make an intelligent guess. Most data compression techniques rely heavily on redundancy of letter and word associations. Over 25 percent of average English text is made up of just ten words — I, is, it, that, the, of, and, to, a, in. Of the other words in common use, many contain standard groups of three letters which are sometimes called trigrams — and, the, tha, ent, ion, for, nde, nce and has. In addition, several letters always occur in combination, for example Q is always followed by U. If these letter combinations are replaced in the text by single characters chosen from the other ASCII codes which are not needed in text programs, it is possible to compress text into a comparatively small space. Attempts to code the data in this way using BASIC are, however, unlikely to be very successful because it is likely that the routines to decode the data will slow the game down to an unacceptable level.

Three dimensional games

As you will recall, there are a few locations in the games listings in this book where you can go up or down into new rooms. The number of occasions where this is possible is so small, that I decided to write them as subroutines rather than trying to create a full three dimensional game. Imagine, however, an adventure based on that famout film *Towering Inferno*, where the object is to escape from the building alive or, perhaps, an adventure set in a large office block. In such circumstances, there would be too many floors to consider writing a two dimensional game, so where do we start when developing such a program?

How do you draw a map of a three dimensional game?
The easiest way of tackling this problem is to draw a separate map of each floor of the building and clearly label any stairs (or other means of moving up or down such as lifts), together with the number of the location reached by going up or down from that position. Once you have done this, converting it into the data for your game should be no more difficult than for a two dimensional game. If, for example, location number 45 is by the stairs on floor number 3, the data line would look something like

```
110 DATA by the emergency exit. A sign reads 'Floor
3'.,32,33,0,41,21,67
```

This line would indicate that movement north would take you to location 32, movement south to location 33, movement east is not possible, movement west takes you to location 41, movement up takes you to location 21 and movement down to 67. Of course, you will need to increase the size of the array holding the map to hold these extra numbers. Thus the second number in S% in the DIM statement will need to be increased from 4 to 6. In addition you will need to change the line which READs this DATA into the arrays so that it too reads 6 items rather than 4. The main difference between a two dimensional and a three dimensional game will be that two extra lines will need to be inserted into the main control loop. These lines will be very similar to the lines used to move the player north, south, east or west. The two lines below can be used in any game where the data has been changed in this way, although the line numbers will need to be changed to suit your own program:

```
200 REM ★★ go up★★
210 IF (B$="up" OR D$="go u") AND S%(P%,5)>0 THEN PRINT
"O.K.":P%=S%(P%,5)
220 IF (B$="do" OR D$="go d") AND S%(P%,6)›0 THEN PRINT
"O.K.":P%=S%(P%,6)
```

These lines don't, of course, print any message if the player attempts to move up or down from a location where this is not possible, although it shouldn't take more than a couple of minutes to rectify this.

Don't forget, though, that you are using far more memory space to store the array holding the map and this will reduce the number of features you can pack into your game. Whether you write a game incorporating these ideas or not is really determined by the plot of the game. If it is set in a multistorey building, then you will probably want to write your game in this way.

Commercial games

Although writing your own adventure is a challenge from which you will get a great deal of pleasure, you can't really enjoy playing a game you've written yourself. After all, you do know the solution! Anyone interested in writing adventures will inevitably want to have a go at playing somebody else's, even if it's just to get a few ideas! The following list contains some of the adventures which are available for Amstrad machines at the time of writing, although, hopefully, many more will be available by the time you read this. Your local stockist should be able to get hold of these games for you, but if not, you should be able to find them advertised in many of the computer magazines.

Level Nine Computing
This company have a large number of adventures available for most of the major home computers.

Colossal Adventure
A suberb version of the game which started it all off. This is not a game to be solved in a hurry and if you've never played an

adventure before, you can be sure of many months of pure enjoyment.

Adventure Quest
An epic adventure which follows in the tradition set by Colossal. This one is my own favourite in the middle earth trilogy, containing some of the most fiendish puzzles to solve.

Dungeon Adventure
The final part of the middle earth trilogy. This game contains many elements of the Dungeons and Dragons theme, although a knowledge of this is not necessary to solve the game.

Lords of Time
This game is, quite simply, superb. In it, you play the part of a time traveller who must enter the old clock and turn the cogs to travel through time to many of the different ages.

Snowball
This game is probably the most impressive adventure you are likely to come across in a long while. It contains over 7000 locations, a fact which impresses me almost as much as the game itself. If you're a science fiction fanatic, then this is the adventure for you!

Return to Eden
This is the follow up to Snowball and, although it contains only 240 locations, compared to Snowball's 7000, is the first graphics adventure from this company. I must admit some disappointment with this game. It is as difficult as any of their others to solve, but they have changed the character set and it is, unfortunately, difficult to read the descriptions on the screen. The graphics, too are disappointing. Not that there is anything wrong with them, just that they take so long to draw, that you've forgotten what you were going to do. Fortunately, they can be turned off when you've seen them once! If you can't wait for a graphics adventure, then this could be just the one for you.

Emerald Isle
The second graphics adventure from this company. This one is a little easier to solve than their other games and should prove to be good introduction to adventures.

Melbourne House

The Hobbit
Originally written for the Sinclair ZX Spectrum, this must be the most famous graphics adventure yet. The graphics in the Amstrad version are second to none and the game sets the standard by which all other graphics adventures will be judged. It is another excellent adventure to cut your teeth on.

Interceptor Software
Interceptor have introduced a range of adventures for the Amstrad in which the graphics show just what the machine is capable of. The standard of the descriptions, however, is not up to the standard set by Level Nine, but that is more than rectified by the speed and quality of the graphics. One other thing in their favour is their cost!

Message from Andromeda
A visually stunning science fiction epic suitable for the novice adventurer.

Forest at World's End
A mythical adventure. Although not all locations have graphics associated with them, those that do are quite simply stunning! It is not a particularly difficult game to solve and should make an ideal first adventure.

Jewels of Babylon
In this game, you play the role of the sole survivor aboard a sailing ship after it has been attacked by pirates. The graphics are again excellent and the game seems a little more difficult to solve than some of their others.

Heroes of Karn
Converted from the Commodore 64, this game has many of the features of the other games in the range. Another good graphics adventure.

Amsoft/Abersoft
Amsoft have released a version of the traditional Classic Adventure, so we are fortunate to have two versions to choose from. Which one you pick is a matter of personal taste. For my money, the 'Level Nine' version has the edge, but only just!

Nemesis
This company has released several adventures for the Amstrad featuring the character of 'Arnold Blackwood — Adventurer Extraordinaire'. These include *The Trial of Arnold Blackwood, Arnold Goes Somewhere Else, The Wise and Fool of Arnold Blackwood* and *New Angelique*: *The Grief Encounter*. All of the games are characterised by a sense of humour and are guaranteed to have you puzzling over some of the strange problems.

Gilsoft
Whilst not an adventure, a book on adventures for the Amstrad which didn't mention *The Quill*, would be failing in its duties! This program takes over where my data filing system leaves off. Using the utility, it is possible to create some superb adventures and can be thoroughly recommended.

The Amstrad has proved to be such a popular computer with adventurers that new games are appearing every day. The list above will, inevitably, be incomplete and if I have missed a game out, it is not intentional!

Playing the game

Most of this book has dealt with writing adventures, but little mention has been made of playing them. Much of what's been said about writing games will be of direct relevance to those who want to play adventures, In addition, playing a few games written by other enthusiasts should give you a few ideas for puzzles and guide you into a new direction of exploration.

To the novice who has never played an adventure previously, exploring the territory in a game can be quite a bewildering experience. If you don't chart your progress by drawing a map as you go along, you may soon find yourself going round in circles, or, even worse, continually being killed in the same place. Drawing maps of games written by others isn't always as easy as it sounds. There are some perverted souls who delight in creating a world where the normal rules of logic don't apply. In such games, you may find yourself going north into a new room, but on going back south again, find yourself in a totally unexpected location. This is fine in a maze, but not in the main part of a game. If I come across too many instances where this happens, I usually give up. Not, I hasten to add, because I can't do it, but because I like games to be logical and finding the solution to depend on my own skill rather that on a chance element. My own approach when mapping out somebody else's adventure is much the same as when mapping my own. The major difference being in the numbering. Each time I enter a new location, I add it to the map, place a brief description along side it and give it a number. Fig. 16.1 shows you how I normally cope with a game where normal logic is not obeyed.

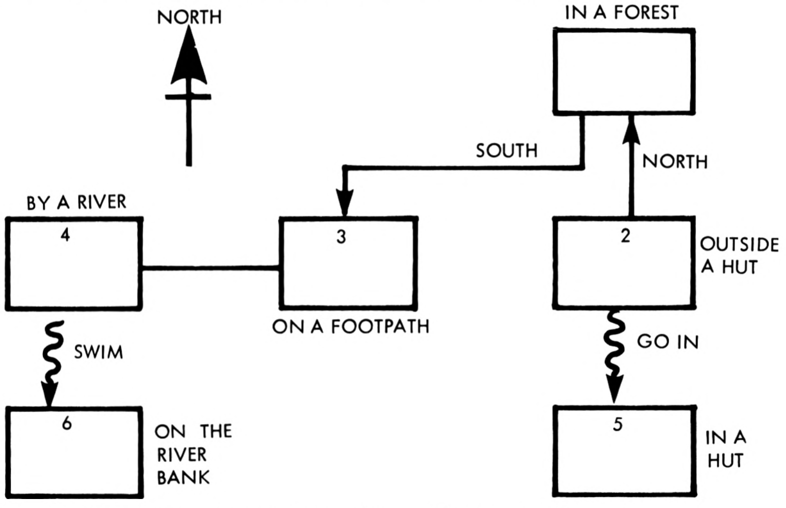


Fig. 16.1 Constructing a map to help solve an adventure game.

How do you draw a map of a maze? This is a question often asked by adventurers and is not an easy question to answer. Until you have actually tried to map your way through a maze, you won't realise just how difficult this can be, especially if the maze is in total darkness! The easiest way of tackling this is to enter the maze carrying as many objects as the game will allow. In each location you enter, try dropping one of the objects so that each time you end up back in that room, you will know exactly how you got there. If the maze is too complex, however, you will soon run out of objects to drop and your map may well end up in the bin. In the original *Colossal Cave*, you were able to carry just enough objects to find your way through the pirate's maze, but in the Level Nine version, you are restricted to only four, making progress that much more difficult.

The secret to a maze actually lies in a password made up of the letters n, s, e, w, u and d (the usual directions), but of unknown length and mixture. Once you have found out the right combination of moves, you will be able to find your way through the maze with no further difficulty, but if you are limited to carrying only three or four items, finding the right combination can be rather more a matter of chance than skill.

In the map shown in Fig. 16.1 the player can move north from location 2 to reach location 1, but movement south takes him from there to location 3. In addition, it is likely that you will come across places where you will need to move by swimming, jumping, crawling, flying or even waiting, adding these to your map can be done by drawing wiggly lines as shown on the map.

As you progress through the game, slaying the odd dragon or two, you are bound to come across the perennial problem faced by all adventurers; that of vocabulary. Many authors are kind enough to provide you with a list of words which the computer understands, or even a full guide to the syntax of the language, as in *The Hobbit*. In most games, however, you are on your own and will have to find out what the computer understands by trial and error. Before starting the game, it's worth spending a little time trying out the 'standard' adventure vocabulary to see which words are recognised.

Does the game allow you to 'get' and 'drop' objects, or does it expect you to try to 'take' and 'leave' them? Some games even give you a choice! Is it possible to 'examine' objects which you are not carrying? Do you have to type in the whole word or will the computer accept just the first few letters ? Typing KIL DRA may annoy the purist, but it is much quicker to type than KILL THE GREEN DRAGON and makes for a quicker route through areas of the adventure which you have previously explored.

One of the most important points to check is whether the computer will allow you to save a partially completed game. This really will save you time later (pardon the pun!), and any commercial game worth its salt will incorporate this facility. To make life easier for yourself, your best course of action is to regularly save a game during play and then when you are much more familiar with the plot, try to save a 'clean' game; that is one where you have achieved as many tasks as possible in the smallest number of moves. This aspect of time is really of vital importance in games such as Colossal Caves where you are carrying a lamp which runs out. If you have a tape with your best performance so far on it, it will allow you to get further into the game without the inconvenience of having to start from the beginning each time.

Playing an adventure game is, in many ways, similar to solving a good crossword — you spend hours puzzling over a clue only to find the solution staring you in the face. All you have to do is to read the messages carefully, think of the obvious (and not so obvious) things to do with the objects you have come across, and suddenly the solution will hit you. Once you know the answer to a puzzle, you'll wonder why you didn't think of it before!

The job of the programmer is to create puzzles which are reasonably devious. Almost anyone can create a puzzle which is impossible to solve, and when you come across a game where your progress is zero, it will soon end up in the bin. A really good game should allow you a fair length of rope and enable you to explore many areas of the fantasy world without being killed in your first few moves.

Yet another point to find out as soon as possible when playing a new adventure is just how many objects you can carry at any one time. Don't forget to look out for those objects in the game which allow you to carry extra items, such as a pack horse, a shopping bag, a tool box or a rucksack. In some games you may come across items which you can wear and this often allows you to carry more. In fact, it may be necessary for you to wear the object in order to complete the game. The palace guards may not let you in unless you are disguised as a soldier by wearing the uniform you found in the cottage, or those alien goggles may allow you to read the strange runes carved into the wall. The magic ring may make you invisible or the rubber gloves insulate you from a nasty shock!

Many of the newer games allow you to enter instrucions in the form of full English sentences such as GO IN AND OPEN THE CUPBOARD DOOR, or TAKE THE RED TOOTHBRUSH AND BRUSH YOUR TEETH. To a newcomer to adventures, this sort of facility does seem to be much more 'user friendly' than the traditional two word sentence input. If that were always the case, however, many of the very best adventures wouldn't be very popular at all!

Once you are fully accustomed to the constraints of two word input, it is surprisingly easy to use and leads to far less confusion by the computer of your exact intentions. If you type in a long sentence and the computer fails to understand the first part of your instructions, it may go on to complete your other instructions, with disastrous effect, or it may just stop at that point. Imagine, for example, that you have typed in KILL THE GREEN DRAGON AND MOVE NORTH, but that you have forgotten to carry the sword needed to kill the dragon. If the computer stops after trying to kill the dragon and tells you that you haven't any weapons, things are not too bad, but if it tries to move you north, then you will very likely get killed. There really is nothing more frustrating than spending three quarters of an hour making good progress in your quest, only to get yourself killed by making a minor mistake. Of course, if you have planned your journey carefully, you will have regularly saved your position on tape or disc so that when you do make a mistake, it is not a major disaster. Don't be surprised, however, if the very first time you forget to save your position, you get yourself killed by an evil Troll! If you are using a tape based system, the time spent in regularly saving your position may seem to be wasted, especially in the more complex games where the routine to save a game may take several minutes, but if you try to avoid it, don't say I didn't warn you!

If you are really determined to succeed and solve the game, it is most important that you read the description of every location very carefully and EXAMINE virtually everything you come across in great detail, as you never know what you might find. Don't assume that the objects you find have to be used in the most obvious way. That screwdriver might be quite sharp on examination and it could make an excellent weapon to stab that evil monster. That piece of driftwood may make an excellent handle for an axe, if only you can find the flint to go with it! Can you open that grate set into the wall, climb the old oak tree, swim across the crocodile infested river or fly on the back of the old eagle?

One trick which is to be found in many adventures is where making an action in one room will have an effect somewhere else in the game. This is sometimes known as the 'Pearl' trick and, if it is used extensively within a game, can lead to confusion. In one recent game, I have come across a large stone wheel and on trying to turn it, have heard a distant rumbling, only to find that I had opened up the snake pit further in my travels. Whenever you do come across puzzles like these in a game, you should try mapping the game both before and after trying out the puzzle to see what difference it really does make.

Yet another method of writing adventures adopted by some programmers is to divide the game up into a number of different sections, each with its own set of problems to be solved before being able to progress to the next section. In a sense, therefore, these games consist of a series of linked mini-adventures, where progress to the next one depends on your completing the previous game. Watch out in these adventures for objects which need to be carried from one location to the next. This is especially true if the game won't allow you to go back to pick up the rope needed to build the raft. In some of the recent games I've played, I've flown on a magic carpet, crashed my plane on a desert island, floated on a log down a river and jumped from a low flying aircraft using a large umbrella as a parachute. There is certainly no going back when the plane has crashed or the carpet flown away!

Finally, watch out for games in which the solution is to be found from a play on words rather than a completely logical approach. The pie man may be a mathematician or the mouse a person. This sort of adventure will have you tearing your hair out at times and is definitely not to be recommended for the beginner.

17 Getting to grips with BASIC

The three adventure listings in this book follow a very similar pattern, based on a set of common subroutines. All adventure games contain a number of standard features and I have attempted to show you how you can adapt one adventure system to deal with several different types of games. This is, however, not the only approach which can be adopted when writing an adventure game. The main advantage of adopting a standard format lies in the ease with which a new game can be created. Once you've mastered the basic ideas, however, you'll be only too eager to experiment with alternative techniques.

Whichever method you choose to adopt, a modular approach does help to make program development much easier and you would be well advised to break your program down into a number of short sections which can be thoroughly tested. Before examining the programming details, however, we really need to take a closer look at some of ideas underlying any adventure. The most interesting and exciting area where different approaches can be useful is in setting problems for the player to solve.

Problems involving objects

Objects in an adventure game can serve several different purposes:

1 Objects which can be seen by the player but not picked up at all, although they may contain a clue written upon them or reveal a second object when searched.
2 Objects which can be seen by the player and readily picked up. These may be 'tools' for use later in the game.
3 Objects which can be seen by the player but can't be picked up until the player has taken some specific action.
4 Objects which can't be seen at first. These may be hidden behind or underneath some other object.
5 Objects which must be moved out of the way to progress further into the game.

6 Objects which are dangerous and which must be eliminated or the player will lose the game.
7 Objects which are described within the main description of the location.

There are a number of advantages to be gained from keeping the description of objects in a separate array from the main description of the location. The main one being that it is easier to write the coding for 'get', 'drop' and 'inventory' routines. That is not, however, to say that it is better from the player's point of view. Games in which there is no attempt to separate the description of objects, directions of motion and locations can be very exciting to play. Writing a program in this way is likely to provide you with a major headache in separating the description of the object from the description of the location and storing it in another array.

Searching through an element of a string array for the object in question is most easily achieved by using the INSTR command. Supposing, for example, that the description of location 21, held in the array loc$(X), is

I am sitting in a chair at the side of the fire. A man with a gun stands by the door and a knife lies on the table.

The player may well try to GET KNIFE and the first thing the program would have to do is to find out if the knife is there. Before being able to do this, the program would, of course, need to split the player's sentence into two words. It may well be that the word KNIFE is held in the variable word$ and the following line could be used to search the description for the word KNIFE.

```
1000 X%=INSTR(loc$(X),word$)
1010 IF X%>0 THEN GOSUB 2000 ELSE PRINT "I can't see it here!"
```

If the word held in the variable word$ is found in the description of the location, X% will store the position in the string where the word is to be found, otherwise it would be zero. Thus line 1010 will print a message that it isn't to be found if X% is still zero. You will, of course, need to write a routine at line 2000 to deal with the action if the object is found in the current location, and this will prove to be quite a complex procedure. The main problem caused by including objects within the main description of the location is in removing it from the description when you pick it up. It's easy enough to remove the word 'knife' from the location, but programming the computer so that it removes the other associated words is going to be very complex.

There are many other possible uses of the INSTR command in an adventure game. One of the features of the method I adopted to analyse the player's instructions is that it is easy to follow. At the same time, however, it does tend to lead to a very long section of coding which in turn is RAM hungry. A far more compact routine can be achieved by storing all the instructions which can be recognised by the computer within one string. This is a technique widely used by adventure writers and is illustrated in the short listing below.

```
100 REM ★★ input instructions ★★
110 INPUT"What do you want to do here";Z$
120 Z$=LEFT$(Z$,3)
130 A=INSTR("EATDRILOOPRAHELSHODROGETPUT",Z$)
140 ON A GOSUB X,Y,Z, ...
```

After the player's instruction is input, the computer looks to see whether the first three letters of Z$ can be found within the string containing words such as EAT, DRINK, LOOK etc. Should a match be found, the variable A will contain a number greater than zero and an appropriate subroutine will be called in line 140.

The routine to split and analyse a sentence is often referred to by programmers as a 'parser', and the parser used here, although efficient in memory usage, is far from tolerant of alternative inputs. The parser, more than any other section of an adventure must be planned very carefully so that the computer recognises the verbs, objects and modifiers you want.

In any adventure, the contents of array elements are being constantly changed and each time the contents are altered the computer will store the new version alongside the old. Eventually the computer's memory will become full and the computer will need to get rid of all redundant messages, objects etc. This process is called garbage collection and will cause the computer to hang up for several seconds. Fortunately, Amstrad computers allow you to clear the contents of an array or variable using the ERASE command:

```
100 ERASE E$
110 IF A=2 THEN E$="East" ELSE IF A=3 THEN E$="West"
```

In this chapter, I have introduced a few ideas which should enable you to explore some of the other methods of doing things. There's a lot to be said for experimenting with various techniques to find the method which suits your own programming style. Whatever method you choose to adopt to deal with standard sections of an adventure such as GO, INVENTORY, GET etc., it's important to make sure that the game is enjoyable to play. Playability can be easily ruined if the parser fails to recognise many of the player's intentions, and for this reason it is good practice to write the game in such a way that the computer understands several alternative words.

Graphics on the Amstrad 18

Amstrad computers boast some of the best graphics around. Of the three modes available, MODE 0 offers the maximum number of colours on screen at once, and although the resolution is not as high in this mode the extra colours mean that you can create more realistic pictures of the locations. Whichever mode you choose, the screen layout will be identical, with 640 locations across by 400 upwards. In the highest resolution mode (2), each location refers to one pixel, whilst in the lower resolution MODE 0, the pixels will be larger.

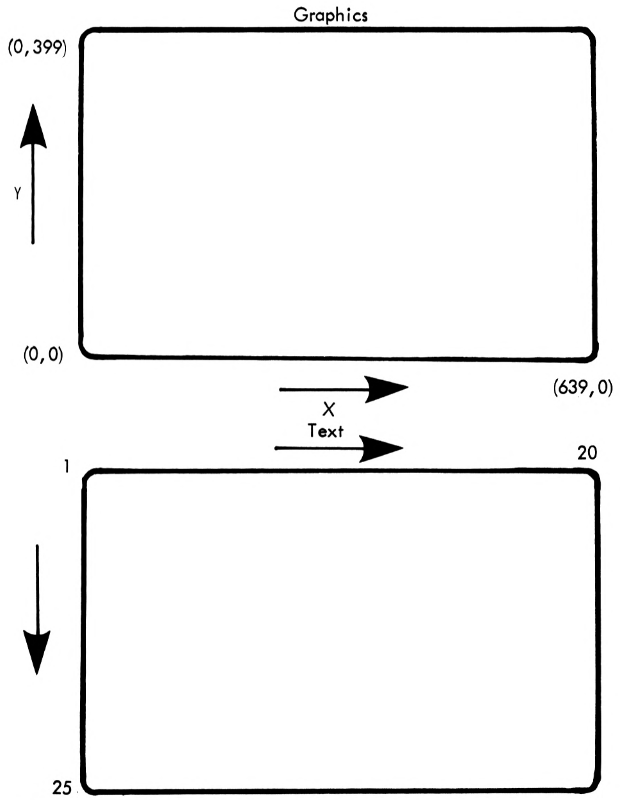


Fig. 18.1 Screen layout in MODE 0.

The 16 default colours available in MODE 0 can be changed using the INK command. If you intend to write a program in which the graphics are to work effectively on both colour and monochrome monitors, you must make sure that the colours used differ in density. It would be little use drawing a picture of a pastel yellow house on a pastel green grass field with a pastel blue roof, as this would show up as one shade on the green screen. The best way of ensuring compatibility is to place dark colours next to pastel colours.

The INK command can be used to redefine any of the PEN/ PAPER colours in the following way.

```
INK X,Y
```

where X= the number of the PEN and Y= the colour chosen from the chart below:

Colour of the INK

0 black	9 green	18 bright green
1 blue	10 cyan	19 sea green
2 bright blue	11 sky blue	20 bright cyan
3 red	12 yellow	21 lime green
4 magenta	13 white	22 pastel green
5 mauve	14 pastel blue	23 pastel cyan
6 bright red	15 orange	24 bright yellow
7 purple	16 pink	25 pastel yellow
8 bright magenta	17 pastel magenta	26 bright white

The important thing to remember is that when the computer is first switched on, the PAPER colour selected will be zero and the pen will be PEN 1. These numbers do not represent the actual colour, but the number of the INK used. The best way to understand this is to imagine a piece of paper and sixteen pens numbered from 0 to 15. The pens can be filled with any of the 27 inks available. All of these pens are initially filled with inks whose colours have been chosen by the designers of the computer. The chart in your computer manual gives the full list of the predefined colours. If at any time you wish to redefine the inks to their default values, the easiest way is to use the call

```
CALL &BC02
```

The colour of the border surrounding the paper can be changed by setting the border to the ink colour desired, although in practice, it isn't usually important to change the border in the pictures for an adventure game. If you want to define one or more of the colours as flashing, you will need to include the number of the two colours after the INK number. e.g. INK 1,26,6 will change PEN number 1 to display flashing bright white/bright red. In most adventures, flashing colours will not be very useful.

In many locations in an adventure game, the picture can be started by drawing the sky and ground and using this as the background on which to build the rest of your picture. The most useful colours for this will be cyan for the sky and a green for the grass. Using just the default colours, this will mean using PEN numbers 8 for the sky and 12 for the grass. There are many ways of filling in large areas, but the simplest is to use two windows. Listing 1 illustrates one way of doing this.

Listing 1

```
10 MODE 0
20 WINDOW #1,1,20,1,10
30 WINDOW #2,1,20,11,25
40 PAPER #1,8:CLS #1
50 PAPER #2,12:CLS #2
```

The window command is used to define an area of the screen on the TEXT axes. Its exact format is

```
WINDOW # stream number,left,right,top,bottom
```

Thus line 20 defines a window at the top of the screen whose paper colour is defined in line 40 and in a similar way, the bottom window is defined in line 30 and the colour in line 50. The window command is probably the simplest way of drawing boxes on the screen, although they may only be placed at the boundaries of the text locations.

Pixel graphics

It is possible to tell the computer to display any of the pixels on the screen using the PLOT command. The general format for doing this is

```
PLOT x,y,col
```

where x = the x coordinate, y = the y coordinate and col = the colour. If you intend to design the picture for a location in the game by addressing each individual pixel, you will need a great deal of patience and will probably start to run into problems with memory shortage. The best way of planning your pictures is to draw a rough sketch of your design on a piece of graph paper which has been labelled with the axes of the screen locations. Programming the computer to draw the picture will be much simpler and less demanding of RAM if you can design it using boxes, triangles, circles or ellipses. It's surprising how a picture built up entirely of these shapes can give a lifelike impression of a location in a game. If you are feeling more ambitious, you can always come back and add the final details to the pictures when you're sure that the game works with memory to spare.

There are a further three commands which are very closely related to PLOT and these are summarised below.

```
MOVE x,y
MOVER x,y
PLOTR x,y,col
```

MOVE is used to move the cursor to the specified pixel on the screen without drawing anything. This is one of the most useful commands available because it tells the computer where on the screen to start drawing the picture. MOVER and PLOTR are very similar to MOVE and PLOT except that they move the cursor relative to the last pixel. If for example, the cursor were at pixel 74,85 and you were to type MOVER 10,0,the cursor would move 10 locations to the right to position 84,85.

Of all the graphics instructions available however, the most useful one of all is the DRAW command. This draws a line from the current cursor position to the new position in the colour specified. Its general form is

```
DRAW x,y,col.
```

Using this command, it's fairly easy to build up boxes, triangles, circles and other assorted shapes.

Drawing boxes

A rectangle or a square is a very useful shape for building up pictures of buildings. The short listing below shows how to build up the outline of a rectangle.

```
5 MODE 0
10 MOVE 10,10
20 DRAW 400,10,1
30 DRAW 400,200
40 DRAW 10,200
50 DRAW 10,10
```

Notice that colour 1 need only be selected in the first DRAW statement unless you wish to change the colour of the subsequent lines. The same effect can be achieved using the DRAWR command as follows:

```
5 MODE 0
10 MOVE 10,10
20 DRAWR 390,0,1
30 DRAWR 0,190
40 DRAWR -390,0
50 DRAWR 0,-190
```

In DRAW statements, where all numbers refer to the actual coordinates, only positive numbers will work, whereas the negative numbers in the DRAWR command imply movement backwards.

Having drawn the outline of the shape, how do you fill it in ? If you own the Amstrad CPC664, you will be able to use the FILL command by moving the cursor inside the box as follows.

```
60 MOVE 200,100
70 FILL 1
```

This will fill the box in with the colour 1. Do make sure, however, that the shape you try to fill is completely enclosed or you will get some very strange effects. For programs which are to be compatible with all Amstrad machines, you'll have to adopt a different approach to drawing and filling in a box. The following listing illustrates one method of drawing a solid box on the screen:

```
10 MODE 0
20 FOR Y=10 TO 200
30 MOVE 100,Y
40 DRAWR 400,0,1
50 NEXT Y
```

Drawing a triangle

Many shapes such as mountains and roofs can be drawn using triangles. The basic outline is easy to draw using just three lines, but filling this shape in on the CPC464 is not as easy as on the 664. The next two listings illustrate different ways of drawing the basic shape of a house. The first listing draws the whole shape in one go, whereas the second draws the roof as a triangle on top of the main box.

Version 1

```
10 MODE 0
20 FOR Y=1 TO 200
30 MOVE 100,Y
40 DRAWR 200,Y,4
50 MOVE 500,Y
```

```
60 DRAWR -200,Y,4
70 NEXT Y
80 MOVE 100,200
90 DRAWR 400,0,5
```

Version 2

```
10 MODE 0
20 FOR Y=1 TO 170
30 MOVE 100,Y
40 DRAWR 350,0,1
50 NEXT Y
60 REM ★★ ROOF ★★
70 FOR Y=171 TO 271
80 MOVE 275,7
90 DRAW Y-110,Y,2
100 MOVE 275,Y
110 DRAW 660-Y,Y,2
120 NEXT Y
```

If you try out the two methods, you'll see that the second method offers the greatest scope in that it allows you to draw the roof in a different colour. The roof is, in fact, built up out of two triangles. Lines 80 to 90 draw the first triangle and lines 100 to 110 draw the second.

Adding windows and doors to your picture of a house is a simple matter of drawing a few rectangles in the appropriate places. Why not try adding a mountain in the distance as well ?

Drawing circles, arcs and ellipses.

These three shapes are very useful to the adventure programmer for drawing cave entrances, railway arches, small hills, the Sun and many other shapes. Unfortunately, Locomotive BASIC doesn't have a CIRCLE command, although the manuals of both CPC464 and CPC664 machines contain routines for drawing circles using the DRAW and PLOT commands (pages F3.12 to F3.14 in the CPC464 manual and pages 58 to 60 in Chapter 1 of the CPC664 manual).

The main problem lies not so much in the routine for drawing a circle, but in knowing how to use it within your own program to obtain the effect you want. Using a circle routine to draw the Sun or Moon is simply a matter of choosing the radius, the centre and colour you want to use and using the routine from your manual. If you want to draw a cave entrance, on the other hand, you will first of all need to draw the floor and the outside of the cave and the simplest way of doing this is to define two windows as discussed previously. All you need to do then is to draw a circle with centre at the middle of the bottom of the screen (320,0).

A small hill can be drawn in a similar way by choosing the centre off the bottom of the screen and selecting one of the green shades for drawing. With a bit of imagination, you'll soon be able to adapt the routine from your manual to help create many curved surfaces. If you're unsure how to set about this, Refer to chapter 9 where you will see how I've used an adaption of the listing in many of the different locations of *Snow White*.

The main process of drawing the picture of a location in an adventure are summarised below:

1 Draw the background (sky and floor).
2 Draw the main foreground objects in layers. Each layer is superimposed on top of the previous one. In this way the house is drawn on top of the background and the windows are drawn on top of the house etc.
3 When you have the framework of your graphics, you should then develop the text for your game.
4 Once the game is running correctly you can come back and add any extra details to the pictures if there is sufficient memory left.

There are several other graphics commands which can be used in Amstrad machines, but many of these are of little use in an adventure. The TEST and TESTR commands for instance allow you to test the colour of a pixel on the screen which is great for arcade games but is of little relevance to an adventure. There are, however, a few games appearing which are a cross between arcade and adventure games and you may well find yourself adapting your game to contain some action sequences.

In a similar way, user defined characters (UDGs) are usually of more use in arcade style games. In *Snow White*, however, I deliberately built up the graphics for the ghost using UDGs to show you how animation can be added to an adventure. Adventures generally don't contain many graphics built up of UDGs because they do take up quite a lot of memory.

Index

ADVENTURE PROGRAMMING ON THE AMSTRAD CPC464 & 664

Would you like to learn how to write adventure programs on your Amstrad micro? Well here is the book to show you how, whether you have a CPC 464 or 664. First it explains how to develop a plot, how to draw a map of the game and how to decide on the locations of the various objects. Steve Lucas then shows you how to translate the game into DATA statements and how to create DATA statements for the objects 'ladder', 'rope', etc. The main control section of any adventure game consists of a control loop which calls the various subroutines, and the book explains how these routines are set up to deal with such features as 'get', 'drop', 'unlock', etc. Throughout this section of the book a full text only adventure game has been developed, which by this time you're probably ready to play!

Amstrad machines have excellent graphics facilities which are both sophisticated and easy to use. The second program, Snow White, shows how these features may be incorporated within your own program to create a game with a full high resolution picture for each location. This game also shows how you can add the final touches to a game using the Amstrad's superb sound features.

The book closes with a look at data files. By saving the data for a game on tape or disc, rather than within the program itself, it becomes much easier to write a completely new game. The final listing is a short program which creates the data file for A Journey Through Space. When it is run, you are given the option of changing the locations and the objects so you can create a game of your own with minimum of fuss.

So there you are all you would-be Amstrad adventurers, here's the book to start you off.

ISBN 0-85242-856-1

9 780852 428566

ADVENTURE PROGRAMMING ON THE AMSTRAD CPC464 & 664

ARGUS BOOKS