ARGUS BOOKS

# MSX

## ADVENTURE PROGRAMMING

STEVE LUCAS

PUBLISHED IN ASSOCIATION WITH MSX User

# MSX ADVENTURE PROGRAMMING

# MSX
## ADVENTURE PROGRAMMING

Steve Lucas

ARGUS BOOKS

Argus Books Limited
1 Golden Square
London W1R 3AB



Series Advisory Editor: Dave Carlos

ISBN 0 85242 857 X



Phototypesetting by Photocomp Ltd., Birmingham
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 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, while others have come as a result of constant refinement of adventure techniques. To the new owner of an MSX 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. MSX BASIC is one of the most sophisticated versions of the BASIC language on the market and 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 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 your 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, while 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 O.K., 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.

The three complete adventure games listed in this book all illustrate different aspects of adventure writing.

| Program | Chapter | Description |
|---|---|---|
| The Wizard's Quest | 1 | A traditional text game set in Middle Earth |
| Snow White | 9 | A full high resolution graphics game for children |
| A Journey through Space | 13 | A science fiction game using data files to make it easy to change |

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 full 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 commands like '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 MSX microcomputers have just over 28K of RAM free for BASIC programs and although the BASIC interpreter is very efficient in its memory requirements, 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 limitations of the language, the response time should still be fast enough for even the most discerning player.

# Programs

**1 The Wizard's Quest** Chapters 1-7

This is a traditional text-only adventure game and 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.

# Contents

# Why MSX? 1

MSX BASIC is a very sophisticated version of the BASIC language, capable of supporting a very wide range of functions. Although most MSX micros sold in Britain have 64K of RAM, only about 28K of this is free for BASIC programs. This does, of course, place a number of limitations on the programmer determined to write a large adventure game. Only a couple of years ago, most computers had only 16K of RAM, or even less, free for BASIC programs. Many very sophisticated adventures were written to run on 16K TRS80's, Commodore Pets and even ZX81s. Since then, however, the average computer owner has come to expect games which include full high resolution graphics, sound effects and full sentence decoding. All of these facilities tend to be very RAM hungry and unless the program is written in machine code, it is unlikely that you will be able to fit graphics, sound and full sentence decoding into an adventure game in most home micros. Where the MSX system scores over many of its competitors is in having a wide range of very powerful graphics and sound commands available in BASIC which allow some splendid pictures to be drawn to illustrate locations in adventure games without using massive amounts of user memory. In addition to the graphics macro language (GML), there is also a comprehensive set of sound commands available which allow the programmer who is prepared to experiment to produce some excellent sound effects.

Many other home computers seem, at first sight, to offer more facilities than MSX machines. The Commodore 64 has more RAM free for BASIC programs, although anyone who has tried to use the graphics or sound facilities of this machine and been put off by the seemingly endless use of the POKE command will be absolutely delighted to find out just how easy it is to produce superb pictures with MSX BASIC. Acorn's two home microcomputers (the BBC micro and the Electron), on the other hand, have graphics and sound commands which are equally powerful, but once the user decides to use a full colour graphics mode, only 8K or so of memory is left to fit the program in. This situation is made even worse when disc drives are fitted. Even the ever popular Sinclair Spectrum can't offer all the facilities available to the MSX owner. As far as graphics

and sound are concerned, the MSX system offers probably the best combination of commands, with plenty of free memory for adventure game writers, of any of the microcomputers yet on the market.

The best tape-based commercial adventures available in Britain are, without doubt, those available for the Sinclair Spectrum, although there are some equally good games appearing for the Commodore 64. In the United States, where disc drives are far more common than they are here, the situation is very different. Some of the very best adventure games of all are available on disc for several different microcomputers, including the Commodore 64 and Apple computers. At the moment, the cost of many of these imported games is extremely high, although Commodore owners will find that some of the games written by INFOCOM have been reduced in price recently.

This will, of course, be of little more than academic interest to the average owner of an MSX micro, eager to get his or her hands on adventure games for their own computer. The MSX system is too new for any original games to have been written to take full advantage of the facilities available and all the adventures I've seen so far have been conversions from programs written for other machines. Not that they should be rejected on these grounds alone, for in fact the programs I've seen so far, including the excellent adventures from Level Nine Computing and the superb version of the Hobbit, really do give the MSX enthusiast something to get his teeth into. What is disappointing, at the moment, is the small number of such games and what better way of rectifying this lack of adventure 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 MSX BASIC really comes into its own for writing adventure games. As you might well expect, MSX BASIC contains all the usual string handling commands such as LEFT$, MID$ and RIGHT$ to allow the programmer to manipulate the text part of the game. In addition, however, there are a number of other facilities available which are not common in home computers. The INSTR command is an extremely useful command which makes writing routines to analyse the player's instructions much easier.

Unlike the BASIC used in many home micros, MSX BASIC is very efficient in its use of arrays. When READing DATA into an array in many other computers, far more memory is used than in MSX machines. Nevertheless, adventure game programmers will probably still want to save every byte of RAM possible so as to pack as many features into the game as possible. There are many ways of saving memory space when writing adventures. Some of these are of universal use, whereas others should be adopted only when you don't intend to have your program published as a listing. The short list below suggests some of the ways of saving memory:

1 Remove all spaces between key words. This method is very effective in saving memory. Unfortunately, however, if a program is printed in a magazine or book which doesn't leave spaces between key words, it is far more difficult for a computer novice to type in without making mistakes. Any magazine editor will tell you of the many letters received from new computer owners complaining about problems encountered with program listings. Most of these difficulties are caused by simple typing errors and it is always good practice to make a program listing as easy to type in as possible if it is to be published as a listing. Just compare the two lines below to see how much easier it is to follow a listing which leaves spaces between key words.

```
10 IF(P%=3ANDSA=7)OR(P%=99ANDAR=3ANDAS=3)OR
(P%=8ANDR=9)THENGOSUB900:PRINT"O.K.":GOSUB800:
RETURN
```

or

```
10 IF (P%=3 AND SA=7) OR (P%=99) AND AR=3 AND AS=3)
OR (P%=8 AND R=9) THEN GOSUB 900:PRINT"O.K.":
GOSUB 800: RETURN
```

There is little doubt that the second version is far more readable and will lead to fewer errors on the part of the typist.

2 Remove all REM statements. These are totally unnecessary and any program will work equally well if they are left out. In this book, however, they are used extensively because they do help to explain what each section of a program is doing.

3 Use integer variables wherever possible. All numeric variables used in adventure games refer to locations or objects and, as it would be impossible to have location 2.7 or object 3.8, you may as well use integer variables. There are two ways of doing this on MSX micros, both of which are very easy to implement. Firstly, you can put a % sign after the variable name, P% instead of P, or you can define certain variables as integers right from the start using the DEFINT command. A little care is needed when using this command, because once a variable name has been defined as an integer, you can't use a string variable with the same name.

As an example of this, you may have used the command:

```
10 DEFINT A-P
```

at the start of your program. After this command has been issued, the computer will interpret all variables beginning with the letter B as being integers and if you then try to use a command referring to B$, you will probably come across problems. There is also a DEFSTR command available which allows you to define variables as being string variables. To illustrate this, consider the following listing.

```
10 DEFSTR P-Z
20 P="In a large meadow."
```

You will notice that it is not necessary to include the $ sign after a variable name if the variable has been defined as a string. I have not used this command in any program in this book as it can lead to confusion!

4 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 in memory for it and if it isn't used, you are wasting RAM. Suppose that you DIMensioned the array A$ at the start of the program with the following line:

```
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.

# Introduction

**Where have adventures come from?**
Although the 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 1970s few people thought that it would be possible to write an adventure game which would run in 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 a microcomputer and went on to produce a now famous game called 'Adventureland' for the Tandy TRS80. It was this game which 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 MSX micros, although I'm sure that if enough people demand MSX 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 Toshiba HX10.

Many software houses have attempted to convert the original 'Colossal Caves' to run on microcomputers and MSX owners are now able to buy one of the best versions around. Level Nine Computing, a British software house famous for their superb adventure games, have managed to cram over 70 extra locations and a new 'end game' into their version of the original game. In order to compress such an enormous amount of data into an MSX micro-computer, 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 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 one 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 one of the few games available for the MSX system.

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 and 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 nothing against such games, although in practice, I have yet to come across a really good implementation of a 'D&D' game on a micro-computer 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. While I would agree that some of the text adventure games are superb, there is 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 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 MSX micros, 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 decide to 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 detail 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 inhabitated 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 well 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 planet to pick up the plutonium he needs for his later mission.

It is well worth while playing as many different adventures games as possible on your MSX computer 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 provide a much clearer conception of what you want to achieve. The very best starting point for any adventure is to sit down with a pad of 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 game.

### 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 have 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 planet 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 villages 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 your 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, whereas 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 than 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, the speed of a game written in BASIC is often very slow, resulting in slow 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 MSX machines, which have a particularly fast version of BASIC. 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 in a 64K MSX micro 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 occupying nearly all of the user RAM of your micro, 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 MSX computer will provide you with a challenge which 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, while 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 MSX computer and this new-found knowledge should encourage you to attempt ever more adventurous programs.

There are already a number of excellent adventure games around for MSX computers, but few of these were originally written for MSX micros and if you have already got the germ of an idea, you'll be only too eager to get started and develop it into a truly original program.

Although there are a few really good adventures for MSX computers, 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 similar categories: thrillers, science fiction, historical, 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. Too many people are ready to criticise your ideas, yet few are prepared to offer constructive suggestions. 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 paperwork. 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 of a nuisance, 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 the flowchart below shows how the game may be broken down into simple, easy to develop stages.

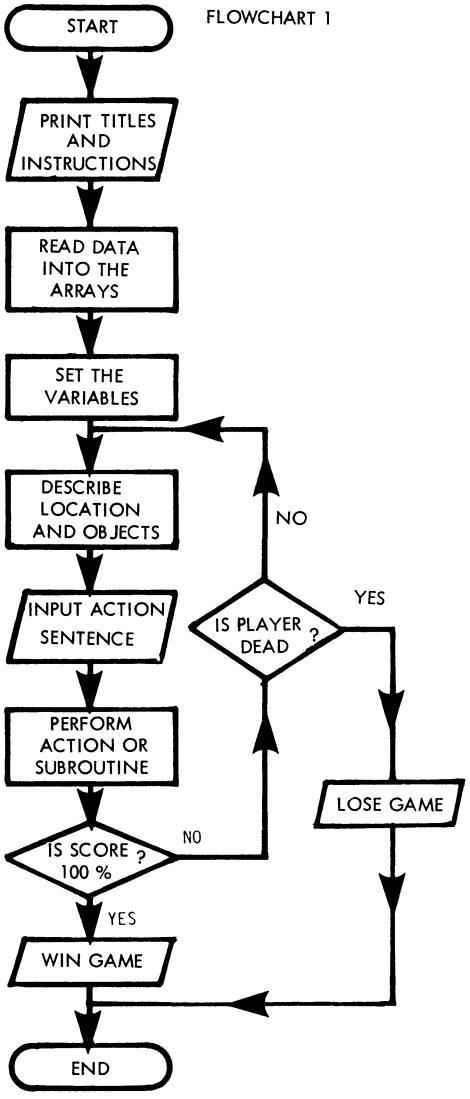



Fig. 1.1 Flowchart for typical adventure game

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 and 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 while considering some of the limitations of the MSX system. Most MSX micros have 64K of RAM, although only 28K of this is available for use in BASIC. When compared with other machines which have recently appeared on the market, this may appear to be rather small. In practice, however, MSX BASIC is so efficient that you will be able to fit more locations into your game than you might imagine. 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. If the save game routine is left out of the program, it runs with over 8K of memory free, but this routine uses nearly all of this space. The program contains many spaces between words to make it easier to follow and if these are left out, you should be able to save a further 2K of 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 graphic 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 11K 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 to fit well over 200 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 on to 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 the illustration on page 18.

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 the prime compass points will use even more RAM. For this reason, I have not included 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!

The diagram below shows one approach to map drawing for adventure writing. Each location is given a discrete number and I've

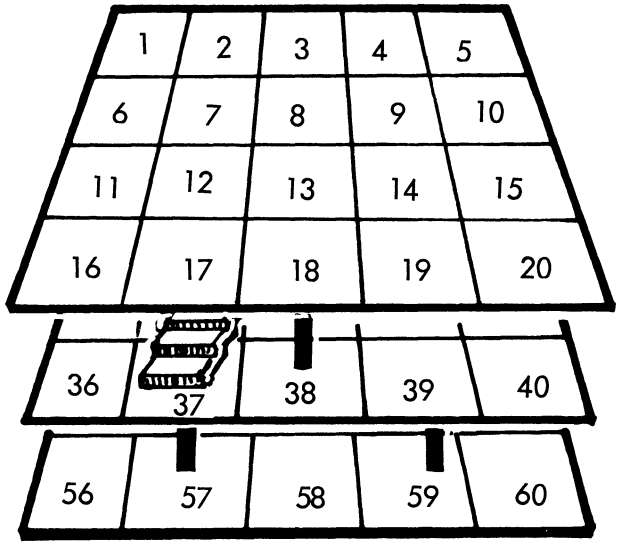



**Fig. 1.2** Map for three dimensional game

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 to 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. The map below shows just 10 locations, but you can add as many rooms to your game as you would like (within reason of course!)

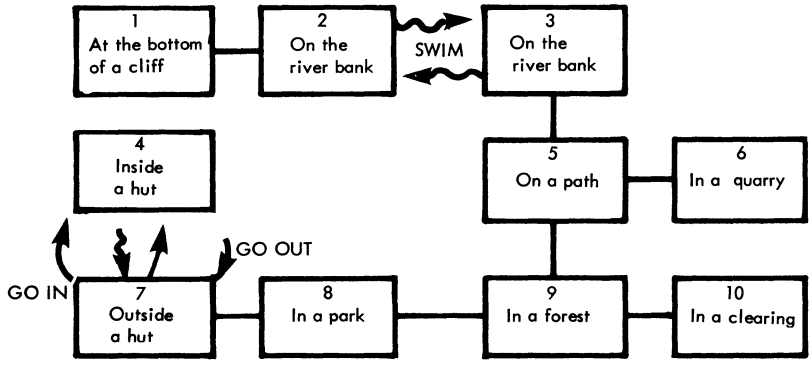



**Fig. 1.3** Drawing a map for an adventure game

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.

## The Wizard's Quest

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 you prefer to solve it on your own, you should jump straight to the listing and type it in.

*The Plot*

Many years ago, in a land far 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 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?

```
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

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. 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 chart below.

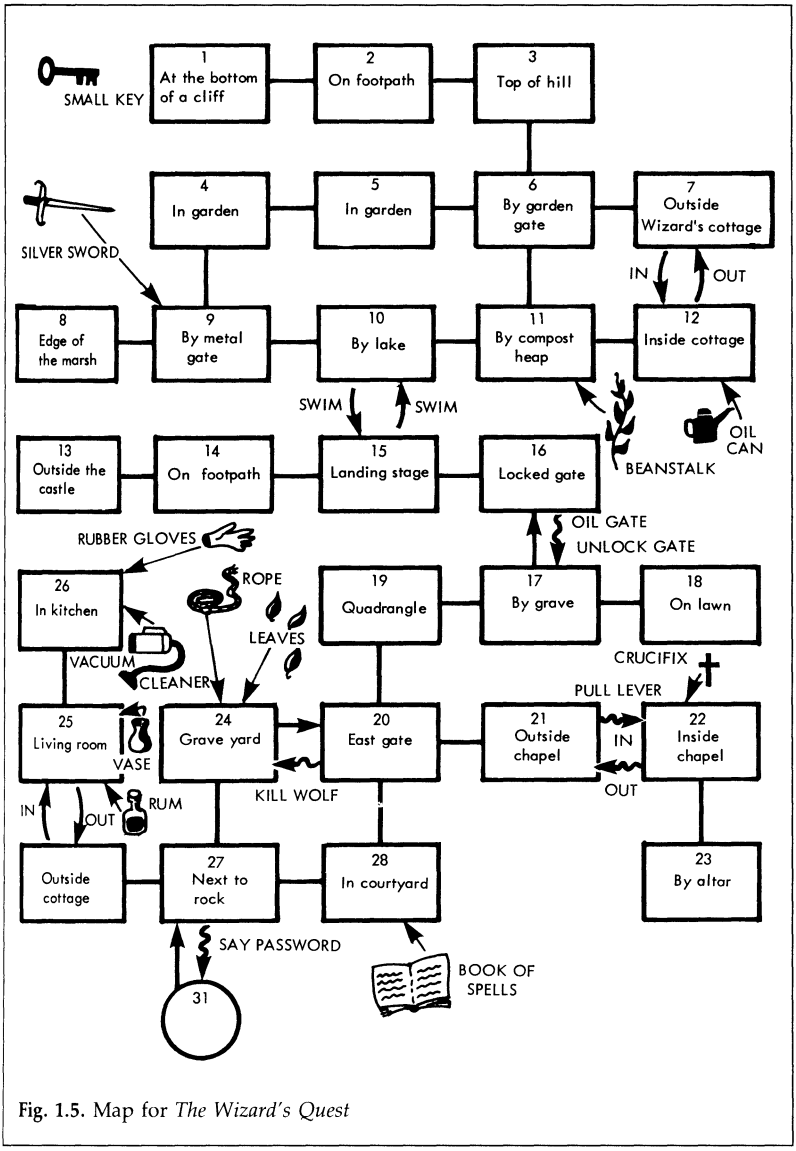
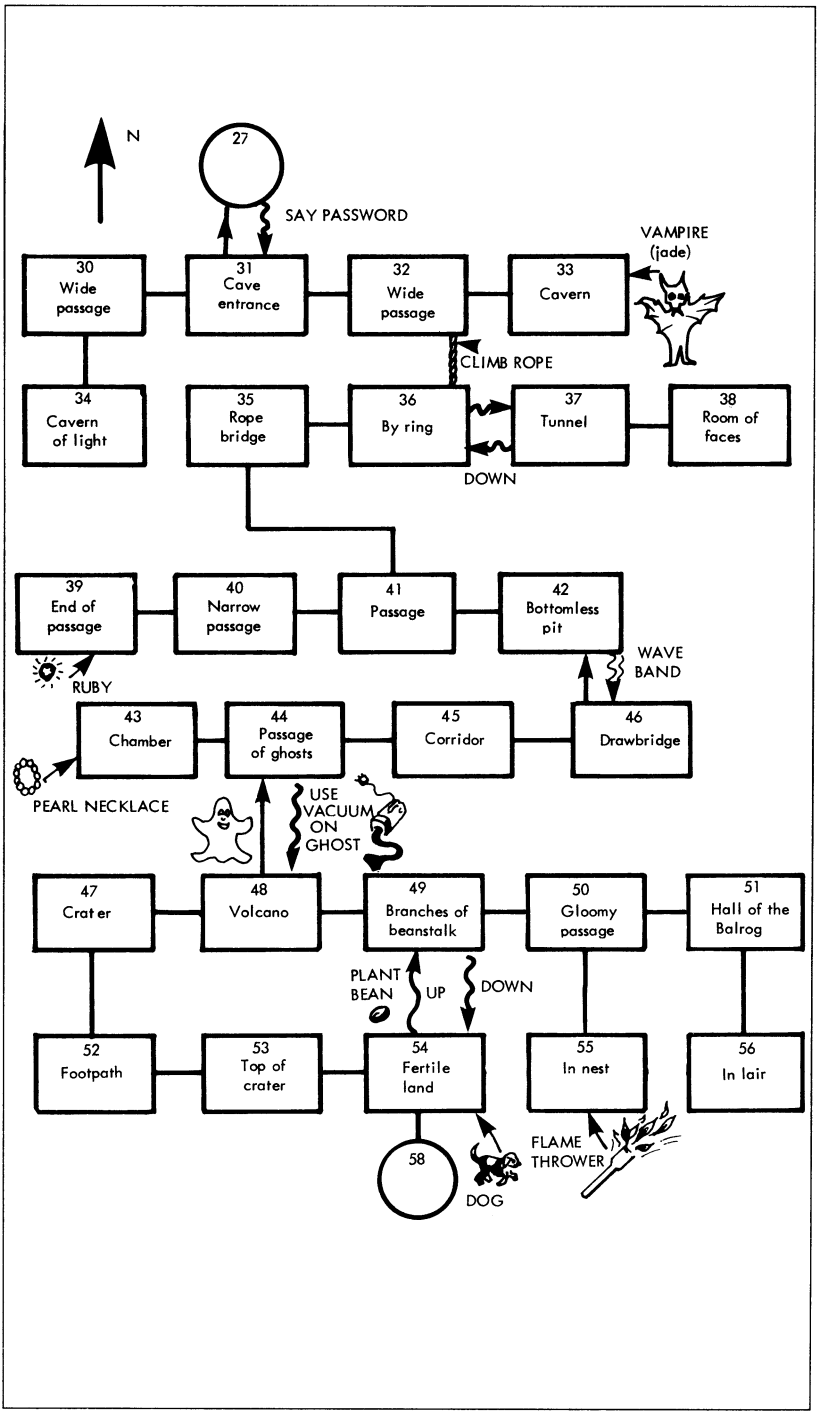
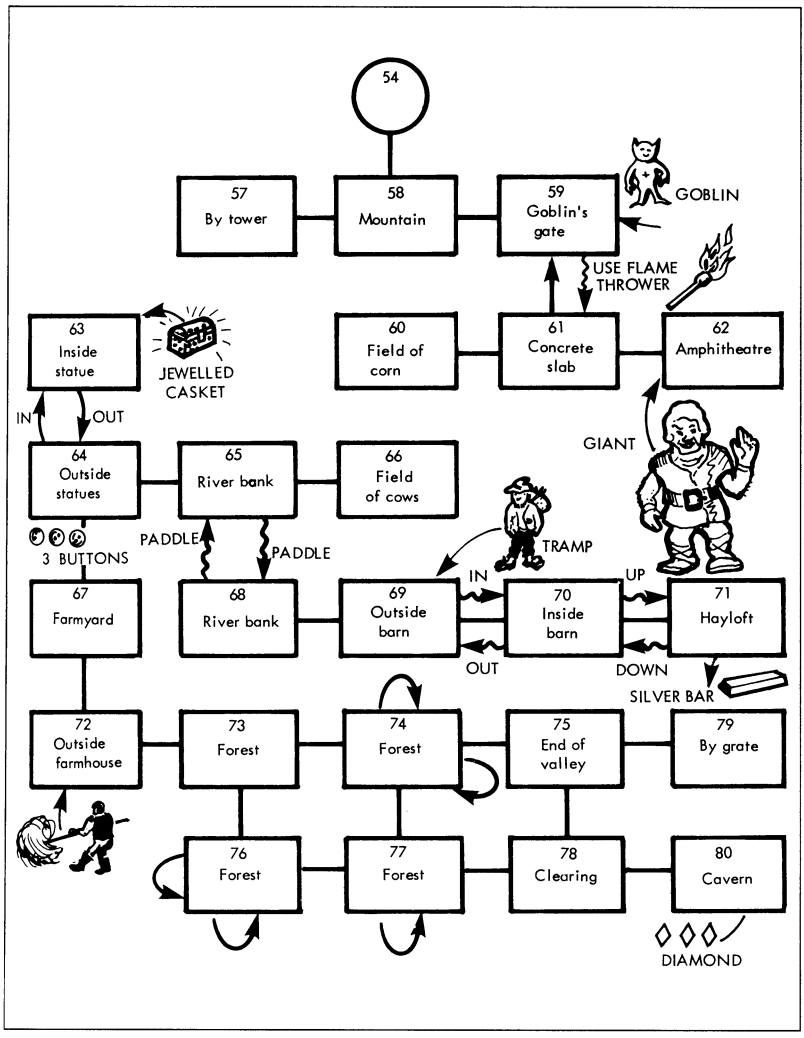



Fig. 1.5. Map for *The Wizard's Quest*


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




| 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 game, 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 bar.

*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 | |

| Number | Original object | Location | Does it change? | What to? |
|---|---|---|---|---|
| 9 | a book of spells | 28 | no | |
| 10 | a gleaming sword | 9 | no | |
| 11 | . . . . . | 24 | yes | a rope & 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 | disappears |
| 27 | . . . . . | 72 | yes | 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.

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 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 descriptions 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 number 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<br>2 unlock it with the key |
| 20 | the wolf blocks your way | kill it with the sword |
| 27 | no way into caves | 1 Read the password<br>2 Say the password |
| 36 | the cave overhead is too high to reach | 1 Throw the rope which will catch on the ring<br>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<br>2 Fill the vase with water<br>3 Pour the water onto the beanstalk to make it grow<br>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 |

| Location | Puzzle | Solution |
|---|---|---|
| 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<br>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 wearing the rubber gloves, the player will end up dead!

In a few locations the player must adopt a different approach to moving in the normal compass 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 the 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 with 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 effect 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 and inserting pictures as an afterthought 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.

# Writing the data 2

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 LOCATION$(7) will immediately remind you of their purpose. Adventure games are, however, very hungry for RAM and using long variable names does use rather more memory than is absolutely necessary. For this reason, I have used single letter variable names for all variables in the games in this book. If you decide to use long variable names, do remember that they must NOT contain keywords. Thus place$ would be valid, whereas locate$ would not!

One point worth noting about MSX BASIC, is that only the first two letters of a variable name are significant. Thus AIR$ and AIS$ will be treated as being exactly the same. Remember also that variable names may be typed in lower case, but all MSX machines will convert them into upper case.

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$(X) | holds the descriptions of the locations |
| G$(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 you are carrying a particular object |

In a text adventure on an MSX machine, the programmer will probably write the game in SCREEN 0. The default setting of the width of this text screen is 37 characters and, in my experience, this is probably the best screen width to use if the game is to be played on a TV set, because the characters at the end of a line tend to disappear off the edge of the screen of a TV set. Users with a monitor will probably want to take advantage of a 40 column screen.

In line 40, I have selected white letters on a blue background and have, in addition, turned off the messages which normally appear at the bottom of the screen about the current function key definitions (line 40).

```
10 REM ** The Wizard"s Quest **
20 REM ** an adventure for MSX micros **
30 REM ** Steve Lucas  1985 **
40 SCREEN 0:KEY OFF:COLOR 15,4
50 LOCATE 8,2:PRINT"The Wizard's Quest"
60 LOCATE 4,10:PRINT"An adventure by S. W. Lucas"
70 WIDTH 37:CLEAR 7000
80 Y$="O.K.":YA$="I can't go that way!":YB$="Don't
 be absurd!"
90 REM ** DIMension arrays **
100 DIM Q$(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 gulley at the bottom ofa sh
eer rock face.,0,0,2,0
160 DATA on a narrow footpath between two highmoun
tains.,0,0,3,1
170 DATA at the top of a small wooded hill. A stee
p footpath leads west.,0,6,0,2
180 DATA on a dirt track which winds its way  thro
ugh a neat garden.,0,9,5,0
190 DATA in a garden full of beautiful flowersand
small trees.,0,0,6,4
200 DATA by a garden gate. The path to the    nort
h leads into open countryside.,3,11,7,5
210 DATA outside a small cottage. A sign on   the
door reads 'Wizard out at the    moment. Please le
ave treasures        inside.'.,0,0,0,6
220 DATA on the edge of a marsh. A sign here  read
s 'Danger do not proceed west!'.,0,0,9,0
230 DATA by a large wooden gate. Strange runesare
inscribed on it!,4,0,10,8
240 DATA on the shores of a small lake. An    isla
nd lies in the middle.,0,0,0,9
250 DATA by the compost heap. Several small   bean
stalks are growing out of it.,6,0,0,0
260 DATA inside the Wizard's cottage. A small fire
 burns in the grate.,0,0,0,0
270 DATA outside a gloomy castle. There seems to b
e no way in.,0,0,14,0
280 DATA on a footpath lined with dense shrubsand
tall trees.,0,0,15,13
290 DATA on a landing stage. A few boats are  moor
ed here.,0,0,16,14
300 DATA at the entrance to a disused         grav
eyard. A rusty chain is padlockedaround the gates.
,0,0,0,15
310 DATA standing next to an old gravestone   whic
h is engraved with the message   'Please Help me!'
,16,0,18,19
320 DATA standing on a small lawn with a tall hedg
e on three sides.,0,0,0,17
330 DATA in a small quadrangle full of ancienttomb
s.,0,20,17,0
340 DATA by the East Gate. A howling wolf     guar
ds the way west.,19,28,21,0
350 DATA outside a small chapel. The door is  clos
ed at the moment. A lever         protrudes from th
e wall.,0,0,0,20
360 DATA inside an ornate chapel. The door hasclos
ed behind me.,0,23,0,0
370 DATA next to the altar.,22,0,0,0
380 DATA outside the graveyard. A path leads  sout
h and down from here.,0,27,20,0
390 DATA in the living room. The woodcutter isasle
ep 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 theway
into a cavern. Strange runes are engraved on it.,2
4,0,0,29
420 DATA in a narrow courtyard which is full  of o
ld bones.,20,0,0,0
430 DATA outside a wooden cottage. A sign on  the
door reads 'Woodcutter for hire'.,0,0,27,0
440 DATA in a wide passage lit by a strange   gree
n glow coming from the south.,0,34,31,0
450 DATA standing in the entrance to the      cave
rns of Xarda.,27,0,32,30
460 DATA in a wide east/west passage. A       smal
ler passage leads south and down from here.,0,36,3
3,31
470 DATA in an enormous cavern lined with     grot
esque faces.,0,0,0,32
480 DATA in the cavern of light.An enormous   crys
tal in the centre sends rays of  green light acros
s the walls.,30,0,0,0
490 DATA on a narrow rope bridge across a deepgull
ey.,0,41,36,0
500 DATA in a small cavern. There is a tunnel high
 above my head leading east. A   metal ring hangs
from the ceiling    next to the tunnel.,32,0,0,35
510 DATA in a gloomy tunnel which looks down  on a
 small cavern. A rope hangs down.,0,0,38,0
520 DATA in the room of many faces. The walls are
lined with mirrors which reflect my image in all d
irections.,0,0,0,37
530 DATA at the end of a passage leading into the
mountain. The view over the      valley is magnifi
cent.,0,0,40,0
540 DATA in a narrow east/west passage lit by dayl
ight.,0,0,41,39
550 DATA n a narrow east/west passage. To the nort
h lies a rope bridge across a    deep gulley.,35,0
,42,40
560 DATA at the edge of a bottomless pit.     Ther
e's a drawbridge on the far side.,0,0,0,41
570 DATA in the dwarf's cavern. It is full of very
 small furniture.,0,0,44,0
580 DATA at the end of a wide passage. An evilghos
t stands guard and won't let me  pass south.,0,0,4
5,43
590 DATA in a wide east/west passage lit by   torc
hes 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 infrom
 an opening to the west.,44,0,0,47
```

```
630 DATA in the branches of a giant beanstalk.Ther
e's a cave entrance to the east.,0,0,50,0
640 DATA at the entrance to a gloomy passage. The
beanstalk prevents much light    entering.,0,55,51
,49
650 DATA in the hall of the evil Balrog. All  the
walls are scorched.,0,56,0,50
660 DATA on a footpath leading between the    cent
re and the top of the crater.,47,0,53,0
670 DATA at the top of of the crater. A path  lead
s east and down the mountainside.,0,0,54,52
680 DATA on a path leading down the mountain. Ther
e is an opening in the cliff     high above my hea
d. The ground here  is very fertile!,0,58,0,53
690 DATA in the nest of the evil Balrog. Threeenor
mous eggs lie at the centre.,50,0,0,0
700 DATA in the Balrog's Lair. A tunnel leads sout
h but the evil stench is too     great for me to p
roceed.,51,0,0,0
710 DATA outside the 'Tower of Darkness'. The entr
ance 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 evilhobg
oblins 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 in frontof t
he 'West Gate'.,59,0,62,60
760 DATA in an amphitheatre. A giant flexes   his
muscles in the far corner.,0,0,0,61
770 DATA inside the bronze statue. A lizard   with
 two heads peers from above.,0,0,0,0
780 DATA on the banks of a river. It is too   dang
erous to cross here. An enormous statue of the god
 'Jolia' lies to thenorth.,0,67,65,0
790 DATA on the banks of the river. It looks  safe
 enough to cross here.,0,0,66,64
800 DATA in a field of grazing cows.,60,0,0,65
810 DATA in a farmyard. A dog sleeps in the   shad
e of the old barn.,64,72,0,0
820 DATA on the banks of a shallow river. A   sign
post here reads 'Danger Quicksandto the west'.,0,0
,69,0
830 DATA outside an old barn. A path leads    west
. An old tramp blocks my way in.,0,0,0,68
840 DATA inside the old barn. A rickety ladderlead
s up into the loft.,0,0,0,0
850 DATA in the hayloft. A cat lies asleep in the
hay.,0,0,0,0
860 DATA outside the farmhouse. It is locked  and
there is no way in. To the west  lies 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 end of a narrow valley.,0,78,79,0
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  lead
s into open countryside.,75,0,0,77
930 DATA at the far end of the valley. A metalgrat
e is set into the ground here.,0,0,0,75
940 DATA in a small hole under the ground.,0,0,0,0
```

Line:

| | |
|---|---|
| 40 | select the text screen, turn off function key messages and select the colours |
| 50-60 | titles |
| 70 | set width of the screen and clear enough string space |
| 80 | define some common messages |
| 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 |

MSX BASIC is very efficient in the way it uses memory to store data, but even so, it will probably be necessary to CLEAR enough string space at the start of the game. This is especially true if you intend to use the routine to load a saved game back into the computer because you will then be storing two copies of the same variable in the memory of your machine. As a starting point, I would suggest that you try CLEAR 1000 and if you get an 'OUT OF STRING SPACE' error, then try increasing the clear command later. As you can see from this game, it proved necessary to clear 7000 bytes of memory, although the game works perfectly with a CLEAR 1000 command provided you don't attempt to load a saved game.

In addition to the set of variables which are standard 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 three common messages, 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 4 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 number 27
and S%(27,4) refers to the location reached by going west from location number 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 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 one*
I am in a gully
I see a key
What shall I do now?

*Example two*
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 28K 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, but 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 number 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 number four is found in line number 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 number 9, whilst movement to the east takes you to location number 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 isn't 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, and if you would like to try your hand at a three-dimensional game, there is more information on the subject in the final chapter.

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.,0,0,20,0,7,19

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 'O.K.' 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 number 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, then this indicates that 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 <RETURN>. 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.

e.g.. supposing the computer prints the value of X as 19, then the last correct location would be number 18, so typing PRINT Q$(17) should print the correct description of location number 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 inverted commas 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 inverted commas, otherwise the computer will interpret the comma as being the start of a new item of data and this will result in a SYNTAX error at some other point in the program.

eg.
250 DATA by a compost heap. A few small, green beanstalks are growing 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 top of the heap.'',6,0,0,0

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 'O.K.', you should type in the short line below and press <RETURN>, This will print out the numbers from 1 to 80 and alongside each will print the description of the appropriate location. Pressing the STOP key will make the computer pause to give you time to read the screen and check the accuracy of the descriptions.

Line to type in:

FOR X=1 TO 80: PRINT X:PRINT Q$(X): NEXT X <RETURN>

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 37 column screen width, the descriptions have been adjusted, by inserting extra blank spaces, so that no word is started after column 34 and that there are no split words. Even professional games sometimes contain errors of this type (I've found one example in the MSX version of the Hobbit!) and they can be very difficult to track down if you leave the task until later.

The final check which must be made 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 stupid!

After entering so much of the program into your computer, you will probably be feeling tired and if you press on with data entry, you will be much more prone to 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 DATA for objects **
960 FOR X=1 TO 30:READ G$(X),B%(X),N$(X):N%(X)=X:N
EXT
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 little 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 by which the computer will recognise it. 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 that you check that the computer is READing the DATA for the objects into the arrays 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 <RETURN>. 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 one), 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. You have now completed the most important part of the program development. Before starting the next section, you would be advised to take a short break so that you come back to the computer refreshed and ready to go.

# 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 in structure you can keep 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 which 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 such as the one on page 45.

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 score & position **
1080 P%=7:S%=0:CLS
1090 REM ** main control loop **
1100 IF S%>9 THEN 1750
1110 PRINT:PRINT"I am :-":PRINT@$(P%)
1120 REM ** check score **
1130 GOSUB 2000
1140 REM ** describe locations **
1150 A$="":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$="" THENA$="nowhere obvious!"
1250 PRINT:PRINT"I can go :-":PRINTA$
1260 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 :-"
1330 PRINTG$(T):E=1:GOTO 1310
1340 Z$="":PRINT:INPUT"What should I do now ";Z$
1350 REM ** analyse input **
1360 B$=LEFT$(Z$,2):C$=LEFT$(Z$,3):D$=LEFT$(Z$,4)
1370 BEEP:CLS
1380 IF C$="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) ELSE IF (B$="n" OR D$="go n") THEN PRI
NTYA$
1420 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 PRI
NTYA$
1430 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 PRI
NTYA$
1440 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 PRI
NTYA$
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 F=FRE(""):IF S%<10 THEN 1100
1750 CLS:LOCATE 9,5:PRINT"W e l l   D o n e"
1760 LOCATE 1,10:PRINT"You have found and recovere
d all the treasures."
1770 PLAY"132dcdedcded"
1780 LOCATE 2,20:PRINT"Goodbye. Thank you for play
ing!":END
```

*Line:*

1080 set the player's starting position 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 examine the first few letters of the input instructions.
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 'kill', 'attack' 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.

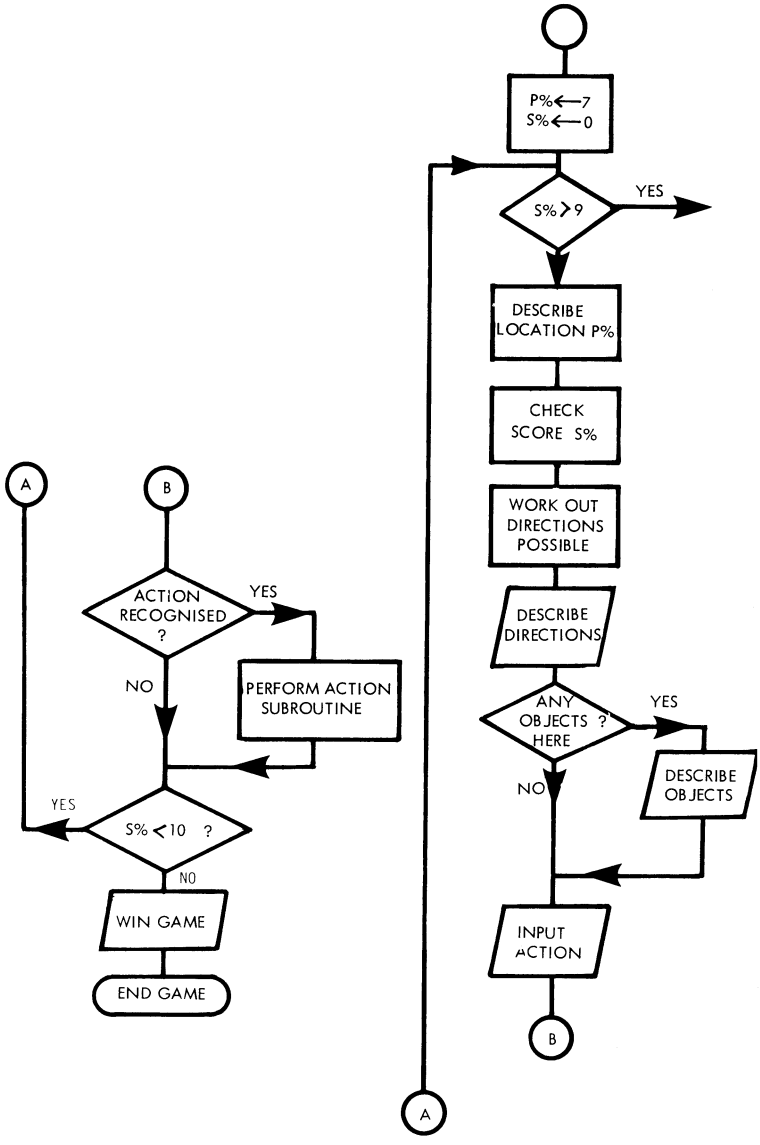



**Fig. 3.1.** Flowchart for control sequence

In all games in this book and, for that matter, in all adventures which I write, the variables 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 1740, 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 his instructions.

You may be wondering what the statement F=FRE("") is doing at the start of line 1740. This instruction is normally used to return the number of bytes of free memory. In this context, it is used for a very different purpose. In any adventure game, the contents of the variables are changed many times during the course of play. In this game, for example, each time the main loop is repeated, the player is asked to type in his instructions. This information is then stored in the variable Z$. Rather than clear out the old contents of Z$, from memory, the computer stores the new variable alongside the old. Thus the first time round the loop, the player may type 'go in', whilst the second time he could type 'get oil'. The computer will not immediately clear the message 'go in' from memory.

Every so often, the area of memory set aside to store variables will become full and the computer, unable to store any new information, will perform what is known as 'Garbage Collection'. This is where it searches through its memory and clears out any redundant words. When it does, the computer will appear to go dead for a few seconds and you may be forgiven for thinking that it has crashed. By putting

the instruction F=FRE("") into line 1740, we force the computer to clear out any old variables every time it goes round the loop and in that way we never allow the area of memory used for variable storage to fill up.

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 1240 checks the directions in which the player 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 number 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 reaching 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 object 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 words 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 familiar two word sentence decoding. It can be extended to provide more complex sentence decoding if you are are prepared to spend some time developing it. The subject of full sentence analysis is discussed in greater detail in the final chapter. In this game, there is little memory free in which to experiment and you would probably need to forego the save game routine and/or shorten the descriptions of the objects and locations.

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 the 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 a 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 convinced that it works correctly, the program can be renumbered to make it easier to follow and also allow space for the next subroutine.

You will notice that the program recognises only lower case instructions. This is a matter of personal choice and you may like to modify the program so that it recognises both upper or lower case commands.

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 and for those who are not quite sure how it works, I'll give a little explanation. 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 element 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.

eg.

```
1370 BEEP: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 your
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 pressed <RETURN>.

Until all of the subroutines have been typed in, you will be unable to test that the main control section works fully, but you 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 CONTROL and STOP keys simultaneously. You can then change your location, without actually playing the game, by typing P%=1 and pressing RETURN. This will move you to location number 1. You can then continue the game by typing CONT and pressing <RETURN>. 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 and 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 players to pit their 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 that 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 them in the right direction and so save them a great deal of needless frustration. This may be achieved by providing a printed instruction sheet to accompany the program or by including instructions within the game.

In 'The Wizard's Quest', there are two subroutines which are not called 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.

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 such foolhardy tricks as jumping from the roof of a burning building or swimming in crocodile-infested water, and 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. Don't worry about the two REM statements, which were put in just to help me to locate my position in the game and which can be left out.

*Subroutine for losing the game*

```
2600 REM ** lose game **
2610 CLS:PRINTE$:LOCATE 0,20:PRINT"Press the <Spac
e Bar> to play again."
2620 A$=INKEY$:IF A$<>" " THEN 2620
2630 REM
2640 REM
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.

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 items **
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 with 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 place, 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.

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 number 12. The ten items of treasure discussed in chapter one 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.

All adventure games need routines which allow the player to pick up and drop objects and the next three routines I want to look at 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 objects **
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 PRINTYB$:RETURN
2170 IF R=12 THEN PRINT"I can't carry them all!":R
ETURN
2180 A(R)=1
2190 E%=0:FOR X=1 TO 4
2200 IF V$(X)="" THEN V$(X)=G$(N%(R)):E%=1:X=5
2210 NEXT:IF E%=0 THEN PRINT"Sorry my hands are fu
ll!":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.

eg.
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 object 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 one and the computer will then check through all 30 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 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 had been set to one respectively. If you can remember back to

chapter one, 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 value 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 one because object number two is the can of oil.

In this game, the player is only allowed to carry 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:PRINT"I am carrying :-"
2310 FOR X=1 TO 4:IF V$(X)<>"" THEN PRINTV$(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 he is carrying, he 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 he has 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 one 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 he drops 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 one.

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 he is.

2340 check the flag to see if the player has filled the vase with water and print the description if he has.

2350 return to the main program control loop.

*The drop routine*

```
2360 REM ** drop item **
2370 GOSUB 2240:IF L%<1 THEN PRINT"I don't have a
";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 hi
m. Here is a gift which may come in useful', he sa
ys.":G$(27)="a sling":B%(26)=0:B%(25)=0:N$(27)="sl
ing"
2440 RETURN
```

Now that the player is able to pick up objects found in the game and the computer is able to tell him which items he is 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 2440 which 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 number 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 call 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 one.
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 with the sections in the next chapter. 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 number 12, which are found in location number 24. The easiest way to test this is to escape from the program using <CONTROL> and <STOP> keys simultaneously, typing P%=24, to change location and then typing CONT before pressing <RETURN>. 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, then 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, move to the 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 move 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 <RETURN>. 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 the 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:PRINTY$:RETURN
1810 IF P%=22 AND SA=0 THEN PRINT"The door's locke
d!":RETURN ELSE IF P%=22 THEN P%=21:PRINTY$:RETURN
1820 IF P%=25 THEN P%=29:PRINTY$:RETURN
1830 IF P%=63 THEN P%=64:PRINTY$:RETURN
1840 IF P%=70 THEN P%=69:PRINTY$: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 MSX BASIC. This means that SA will start at zero and in the PRAY routine this will be set to one. 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 you are in location 12, change the value of P% to 7 and return to the main loop.

1810 if you are 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 you are in location 25, move to location 29 by changing the value of P% and return to the main loop.

1830 if you are in location 63, change the value of P% to 64, print the message and return.

1840 if you are 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 <CONTROL> and <STOP> keys. 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 number 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 PRINTY$
1880 PLAY"l64decdedecdede"
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 do
or 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 one 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 closes 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 you are 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:PRINTY$:RETURN
1940 IF P%=21 AND SB=0 THEN PRINT"The door's locke
d!":RETURN ELSE IF P%=21 THEN P%=22:PRINTY$:RETURN
1950 IF P%=29 THEN P%=25:PRINTY$:RETURN
1960 IF P%=64 AND SC=0 THEN PRINT"The way in is cl
osed!":RETURN ELSE IF P%=64 THEN PRINTY$:P%=63:RET
URN
1970 IF P%=69 AND SH=1 THEN P%=70:PRINTY$:RETURN E
LSE 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 he has to solve certain problems first. In location 21, he cannot go into the chapel unless the door is opened by pulling the lever. This, in turn, will result in death unless he is wearing the rubber gloves to prevent him 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 he has 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 him that he 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 one. 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 you are in location 7, move to location 12, print the message and return to the main loop.

1940 if you are 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 you are in location 29, move to location 25, print the message and return to the main loop.

1960 if you are 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 you are in location 69 and the flag is one, move to location 70, otherwise print the message about the tramp not allowing you 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:PRINTY$:RETURN
```

When the player tries to pull the lever outside the chapel, he will get an electric shock and die unless he is wearing the rubber gloves. If you check through the list of objects, you will see that object number 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', the 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 two, print the message about already wearing them and return to the main loop.

2500 set the value of the flag to two, 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 PRINTYB$:RETURN
2530 IF A(6)<>2 THEN E$="a violent electrical curr
ent surges  through my body. I am dead!":GOSUB 261
0
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 he doesn'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 equal to two 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 PRINTYB$: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 number 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 you 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 THENPRINT"I have nothing to read!":
RETURN
2680 PRINT"Most of the book is in a strange     la
nguage, but one sentence reads, 'Toenter the caves
, 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 number 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 and 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 you are carrying anything to read and print the message before returning to the main loop if you aren'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 ** say **
2710 INPUT"What do you want to say ";Z$
2720 IFP%<>27 THEN PRINTY$;" nothing happens.":RET
URN
2730 IF Z$<>"sdfda" THEN PRINT"nothing happened!":
RETURN
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 cavern's open!":
RETURN
```

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 he wants 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 postion 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 would be suicide!":RETURN
```

```
2790 IF LEFT$(L$,3)<>"wol" OR P%<>20 THEN PRINTYB$
:RETURN
2800 IF A(10)=0 THEN E$="The wolf attacks me first
!":GOSUB 2610
2810 PRINT"The wolf dies!":S%(20,4)=24:Q$(20)=LEFT
$(Q$(20),18):RETURN
```

In this game, the player must kill the wolf, which is found in location number 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, the 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 number 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)="rop
e":RETURN ELSE PRINT"I see nothing!":RETURN
```

If you can remember back to the planning stages 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 you are 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. You 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 one (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 you are 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.

# 6 Setting the puzzles part 3

### Climb

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

The first line of this subroutine calls the subroutine to split the input sentence into two parts. Line 2490 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 you are 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 you are 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 you are 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:PRINTY$:RETURN
3020 IF P%=80 THEN P%=79:PRINTY$:RETURN
3030 IF P%=36 AND SG=0 THENPRINT"not yet!":RETURN
ELSE IF P%=36 THEN P%=37:PRINTY$:RETURN
3040 IF P%=54 AND SL=1 THEN P%=49:PRINTY$: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, while 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 you are in location 70, move to location 71, print the message and return.
3020 if you are in location 80, move to location 79, print the message and return.
3030 if you are 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 you are 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    stup
or, fall 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 number 8. The first line of the routine checks whether the player has tried to drink anything else and prints a message about his stupidity if he has. 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 he trips up in a drunken stupor and dies! 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$;" away!":RETURN
3120 IF A(8)=0 THEN PRINT"I don't have any!":RETUR
N
3130 IF P%<>69 THEN PRINT"There's nobody here who
would like it":RETURN
3140 A(8)=0:FOR X=1 TO 4:IF V$(X)=G$(8) THEN V$(X)
=""
3150 NEXT:PRINT"The tramp thanks me and walks away
!"
3160 Q$(69)=LEFT$(Q$(69),42):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

makes sure 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 number 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$(44),29):RETUR
N
3220 IF R=14 AND A(14)=0 THEN PRINT"I don't have i
t!":RETURN
3230 IF R=14 AND P%<>33 THEN PRINT"There isn't muc
h point here!":RETURN
3240 IF R=14 THEN PRINT"The vampire flees, leaving
 something behind!":SI=1:G$(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 here!":RETURN
3270 IF R=16 THEN PRINT"The slug shrivels to nothi
ng 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$="it explodes and covers me with
 a jet of flames!":GOSUB 2610
3300 IF R=22 AND P%<>59 THEN PRINT"There isn't muc
h point here!":RETURN
3310 IF R=22 THEN PRINT"The flames drive them away
!":S%(59,2)=61:Q$(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"There isn't muc
h point here!":RETURN
3340 IF R=23 AND SM=0 THEN PRINT"The grate opens!"
:SM=1:Q$(79)=LEFT$(Q$(79),30)+"There's a hole in t
he ground.":RETURN
3350 IF R=23 THEN PRINT"It's already open!":RETURN
3360 IF R=27 AND A(27)=0 THEN PRINT"I don't have i
t!":RETURN
3370 IF R=27 AND P%<>62 THEN PRINT"The sling is of
 litle use here!":RETURN
3380 IF R=27 AND SN=1 THEN PRINT"I can't use it tw
ice!":RETURN
3390 IF R=27 AND SN=0 THEN SN=1:PRINT"That's done
the trick! The giant     fades away and I can see
something!":G$(21)="an ** EMERALD **":N$(21)="emer
ald":RETURN
3400 PRINT"I can't use ";L$;" here.":RETURN
```

This is a very useful 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 they 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*
*Lines:*
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 if 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 if 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 if not.
3270 print the message, set 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 you are 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 already 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 you have already used the sling.

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 they 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 and 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:PRINTY$:RETURN
3430 IF P%=15 THEN P%=10:PRINTY$: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 stu
pid 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 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 you are in location 10, move to location 15 by changing the value of P%, print the message and return to the main loop.
3430 if you are in location 15, move to location 10, print the message and return.
3440 if you are in location 65 or 68, print the message and return to the main loop.
3450 if you are 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:PRINTY$:RETURN
4160 IF P%=68 THEN P%=65:PRINTY$:RETURN
4170 IF P%=10 OR P%=64 OR P%=15 THEN PRINT"The wat
er's too deep!":RETURN
4180 PRINTYB$: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 you are in location 65, move to location 68, print the message and return to the main loop.

4160 if you are in location 68, move to location 65, print the message and return to the main loop.

4170 if you are in a 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 PRINTY$:PRINT"The chain comes l
oose.":Q$(16)=LEFT$(Q$(16),47):S%(16,2)=17:RETURN
3510 IF P%=79 THEN PRINT"There's no keyhole!":RETU
RN
3520 PRINTYB$: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 you are carrying the key and return to the main loop if not.

3490 if you are 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 you are in location 16, print the message, change the description of the location and return to the main loop.

3510 if you are 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 THENPRINT"no oil!":RETURN
3560 PRINTY$:SK=1:RETURN
```

Before you are able to 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 the player 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. On route to location 16, you must get the key and when you reach the gate, try to unlock it. Once you have unlocked it, going south from location 16 should take you 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 you are not in location 16, print the message and return to the main loop.

3550 check to make sure that you are 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$(X)="":PRI
NTY$
3610 NEXT:G$(1)="a tiny little beanstalk murmuring
 ...water, water!":B%(1)=54:N$(1)="":A(1)=2:RETURN
```

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 you are 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 the main loop if you are 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%=68
THEN PRINTY$:A(5)=2:RETURN ELSE PRINT"I can't do t
hat here!":RETURN
```

When you have planted the beanstalk, it will start murmuring 'water, water' and your 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 you are carrying the vase 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 one if it is being carried and therefore once it has been filled with water, its value is set to two. Be careful that you don't drop it once it has been filled, or 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 PRINTY$:A(5)=1
3680 IF P%<>54 OR A(1)<>2 THEN RETURN
3690 IF SL=0 THEN PRINT"The beanstalk spursts into
 rapid growth.":G$(1)="an enormous beanstalk reach
ing 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 you are are 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 one.
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:PRINTY$:RETURN
3730 IF P%=49 THEN P%=54:PRINTY$:RETURN
3740 IF P%=71 THEN P%=70:PRINTY$:RETURN
3750 IF P%=79 AND SM=0 THEN PRINT"I can't get past
 the grate!":RETURN
3760 IF P%=79 THEN P%=80:PRINTY$: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 number 36 and return to the main loop.

3730 if the current location is number 49, change it to number 54 and return to the main loop.

3740 if the current location is number 71, move to location 70 and return to the main loop.

3750 if the location is number 79 and the grate is closed (SM=0), then print the message and return to the main loop.

3760 if the location is number 79, the grate must be open, so move to location number 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$
3830 Z$=LEFT$(Z$,1):IF SC=1 THEN E$="A snake crawl
s from behind the       buttons and sinks its fang
```

```
s into 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
```

When you reach location 64, you 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 you need to press anything.

*Line:*

3790 check the location and return to the main loop if it isn't number 64.

3800-3810 describe the colours of the three buttons to the player.

3820 input the player's choice.

3830 if you have already pressed the correct button, the snake will attack you and the lose game routine is called.

3840 if you don'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 one 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 you.

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"Press <Space Bar> when you have set  th
e tape recorder ready to record."
3880 AA$=INKEY$:IF AA$<>" " THEN 3880OK
3890 OPEN "cas:data" FOR OUTPUT AS #1
3900 FOR X=1 TO 80:PRINT#1,Q$(X):NEXT
3910 FOR X=1 TO 80:FOR Y=1 TO 4:PRINT#1,S%(X,Y):NE
XT Y,X
3920 FOR X=1 TO 30:PRINT#1,G$(X):NEXT
3930 FOR X=1 TO 30:PRINT#1,B%(X):NEXT
3940 FOR X=1 TO 30:PRINT#1,N$(X):NEXT
3950 FOR X=1 TO 30:PRINT#1,N%(X):NEXT
```

```
3960 FOR X=0 TO 30:PRINT#1,A(X):NEXT
3970 FOR X=1 TO 4:PRINT#1,V$(X):NEXT
3980 PRINT#1,SA,SB,SC,SD,SE,SF,SG,SH,SI,SJ,SK,SL,S
M,SN,SO,SP,P%
3990 CLOSE:RETURN
```

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 wait for the space bar to be pressed.
3890 open a cassette file as channel 1.
3900 write the descriptions of the 80 locations onto the tape.
3910 write the details of the map onto the tape.
3920 write the descriptions of the objects onto tape.
3930 write the locations where the objects can be found onto tape.
3940 write the words recognised by the computer onto tape.
3950 write the pointers to the words onto tape.
3960 write the flags for the objects being carried onto tape.
3970 write the descriptions of the objects being carried onto tape.
3980 write the flags SA – SP and the current position (P%) onto tape.
3990 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**

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.

```
4000 REM ** load game **
4010 PRINT"Press <Space Bar> when you have set  th
e tape recorder ready to play."
4020 AA$=INKEY$:IF AA$<>" " THEN 4020OK
```

```
4030 OPEN "cas:data" FOR INPUT AS #1
4040 FOR X=1 TO 80:INPUT#1,Q$(X):NEXT
4050 FOR X=1 TO 80:FOR Y=1 TO 4:INPUT#1,S%(X,Y):NE
XT Y,X
4060 FOR X=1 TO 30:INPUT#1,G$(X):NEXT
4070 FOR X=1 TO 30:INPUT#1,B%(X):NEXT
4080 FOR X=1 TO 30:INPUT#1,N$(X):NEXT
4090 FOR X=1 TO 30:INPUT#1,N%(X):NEXT
4100 FOR X=0 TO 30:INPUT#1,A(X):NEXT
4110 FOR X=1 TO 4:INPUT#1,V$(X):NEXT
4120 INPUT#1,SA,SB,SC,SD,SE,SF,SG,SH,SI,SJ,SK,SL,S
M,SN,SO,SP,P%
4130 CLOSE:RETURN
```

*Line:*
4010 print the message asking the player to insert the tape.
4020 wait for the space bar to be pressed.
4030 open channel one to input the cassette file.
4040 input the descriptions of the 80 locations.
4050 input the data for the map.
4060 input the descriptions of the 30 objects.
4070 input the positions where the objects are to be found.
4080 input the words understood.
4090 input the pointers to the words recognised.
4100 input the flags to tell the computer which items are being carried.
4110 input the descriptions of the items being carried.
4120 input the flags SA – SP and the current location (P%).
4130 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 when the OPEN command uses a different file name. 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 listings 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.

One of the problems with using this technique to save the data on an MSX micro is that the variables are given two different values. Thus the array Q$(X) holding the descriptions of the locations has its values held in DATA lines and in addition, extra memory space must be set aside to hold the alternative description which is loaded from tape. This means that we have to CLEAR plenty of space to hold the string variables (7K in this game) at the start of the game (line 70).

In effect, therefore, we are wasting a vast amount of memory space and would be better advised loading our data in from tape or disc every time. This leaves far more room in the game to hold more locations and/or puzzles. The main disadvantage of not including the data within data lines is that every time you make a mistake when developing the game, you will have to load the data tape in again. This in turn means that the program will take far longer to develop, especially as you will also need to write another program to create the very first data file.

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 isn't much spare RAM available after the save game routine has been introduced, even in a 64K MSX machine, but there should be enough room to add a few extra puzzles using some of the objects which, so far, 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"Original string :-"
70 PRINT A$
80 PRINT:PRINT
90 PRINT"Final string :-"
100 PRINT B$
```

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 examining 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 9 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 compressed 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.

# 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 a adventure game aimed at young children and contains a full high resolution picture of each location visited. I have also included a number of sound effects within this game and these too were decided upon before beginning programming rather than adding them 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 in the descriptions fairly short and to make the pictures look as sophisticated as possible. Despite the very advanced commands available in MSX BASIC to help with graphic design, the pictures used in this program do take up a vast amount of user memory. For this reason, I decided to design a game with only 24 locations. In many graphic adventures, including 'The Hobbit', the programmers have chosen not to include a picture of every location. This does allow them to fit in more locations, but even in 'The Hobbit', there are only just over double the number of locations found in this game. You must decide which path to follow right from the start. My own feeling is that a game designed for younger children needs 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 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 the data lines, as you did in the last program, and then develop the graphics, or you can go straight to the graphics. My own preference is to design the graphics first. There is no real advantage to be gained from developing the graphics first, except that you will have a better idea of the amount of memory left for the rest of the game once the pictures have all been completed.

In this game, I decided to write the text in SCREEN 0, whilst the graphics MUST be written in SCREEN 2. The main advantage of changing SCREENs on an MSX micro is that the number of characters which can be fitted across SCREEN 0 is greater than on SCREEN 2.

In addition, text cannot be printed directly onto the high resolution SCREEN unless channel 1 is first opened. This will be discussed later.

In location 22, where there is some animation, I have defined the ghost as a large (16 × 16) sprite and therefore, the graphics must be drawn in SCREEN 2,2,0. The second number following the SCREEN

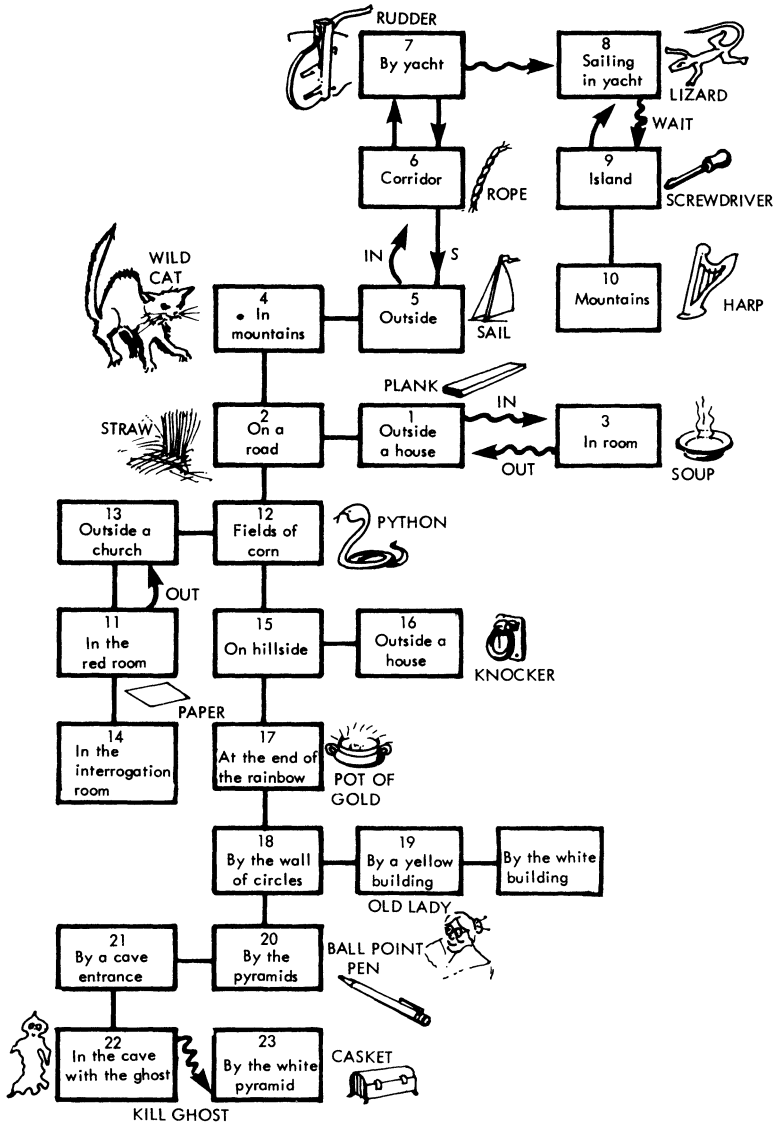



**Fig. 9.1** The map for *Snow White*

command selects the option of using large sprites. When I introduce the flowchart for this game, you will notice that the program is constantly changing from SCREEN 0 to SCREEN 2,2,0. In order to make life easier, I have directed the entry to the graphics section through a subroutine which controls the display.

### Graphics control subroutine

```
2160 SCREEN 2,2,0:OPEN"grp:" FOR OUTPUT AS #1
2170 ON P% GOSUB 2260,2330,2380,2440,2500,2540,260
0,2670,2710,2750,2810,2860,2910,2960,3000,3050,310
0,3140,3200,3260,3310,3350,3430,3470
2180 CLOSE
2190 RETURN
2200 END
```

*Line:*

2160 selects the high resolution screen and opens channel 1 so as to allow text to be printed on the graphics screen.

2170 examines the value of P%, the current location, and calls the correct graphics subroutine.

2180 closes channel 1

2190 returns to the main program.

Whenever this subroutine is called, the program examines the value of P% and calls the subroutine for that particular location. Thus if P%=5 when the routine is called, line 2170 will call the fifth subroutine (at line 2500) to draw the graphics for location 5. Each time you type in the graphics for a new location, you should check it to make sure that it works properly. If, for example, you have just typed in the graphics for location 7, you should follow the following procedure.

1 Type P%=7 press <RETURN>
2 Type GOSUB 2160 press <RETURN>

Line 2170 will then call the subroutine at line 2600, which deals with the graphics for location 7 and you should see the picture connected with it.

Before you reach this stage, however, it is very important that you type in the short listing below. Whenever an MSX micro reaches the end of graphics instructions, it will automatically return to SCREEN 0 and therefore we must include a routine which halts the program until we have viewed the picture.

### Preventing return to SCREEN 0

```
2210 LINE(0,171)-(255,191),1,BF:PRESET(10,181),1:P
RINT#1,"Press <Space Bar> to continue."
2220 F$=INKEY$:IF F$<>" " THEN 2220
2230 RETURN
```

*Line:*
2210 draws a black box at the bottom of the screen and prints the message in it.
2220 waits for space bar to be pressed.
2230 returns to graphics control subroutine.

Note that you must use PRINT #1 to print text onto the high resolution screen and that this only works if channel one has been opened as a graphics channel (see line 2160).

Now that you have typed in the two sections of code needed to control the graphics sections, you can press ahead with the actual graphics. I have split the graphics up so that each location has its own separate listing. If you type in the graphics for each location separtely and test it out, it should make it easier to debug than if you type in the whole lot in one go.

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.

**Graphics commands used**

| | |
|---|---|
| LINE (X1,Y1)–(X2,Y2),C | Draws a line from location X1,Y1 to location X2,Y2 in colour C. |
| LINE (X1,Y1)–(X2,Y2),C,B | Draws a box between location X1,Y1 and X2,Y2 in colour C. |
| LINE (X1,Y1)–(X2,Y2),C,BF | Draws a box and fills it in. |
| CIRCLE (X,Y),R,C | Draws a circle centre X,Y of radius R and colour C. |
| PAINT (X,Y),C | fills in a colour up to a predefined border. |

*Location 1. Outside the small house*

```
2250 REM ** location 1 **
2260 CLS:LINE(0,0)-(255,191),5,BF:LINE(0,90)-(255,
191),14,BF:LINE(150,170)-(240,100),11,BF
2270 LINE(140,100)-(250,100),6:LINE(195,75)-(250,1
00),6:LINE(195,75)-(140,100),6:PAINT(190,80),6
2280 LINE(200,170)-(210,150),1,BF:LINE(220,155)-(2
32,145),1,BF:LINE(220,110)-(232,120),1,BF
2290 LINE(160,155)-(190,145),1,BF
2300 CIRCLE(20,50),15,10:PAINT(20,50),10:LINE(220,
80)-(230,95),12,BF
2310 GOSUB 2210:RETURN
```

*Location 2. On the wide road*

```
2320 REM ** location 2 **
2330 CLS:LINE(0,0)-(255,191),5,BF:LINE(0,70)-(255,
191),14,BF:LINE(80,191)-(120,70),7:LINE(135,70)-(1
```

```
75,255),7:LINE(120,70)-(135,70),7:PAINT(100,170),7
2340 CIRCLE(20,50),15,10:PAINT(20,50),10:LINE(129,
70)-(140,255),1
2350 LINE(200,70)-(250,50),13,BF:LINE(195,50)-(255
,50),6:LINE(225,35)-(255,50),6:LINE(195,50)-(225,3
5),6:PAINT(220,49),6
2360 GOSUB 2210:RETURN
```

*Location 3. In the small room*

```
2370 REM ** location 3 **
2380 CLS:LINE(0,0)-(255,191),14,BF:LINE(0,0)-(50,5
0),11:LINE(50,50)-(50,149),11:LINE(50,149)-(0,191)
,11:PAINT(20,90),11
2390 LINE(50,50)-(205,145),13,BF:LINE(205,50)-(205
,145),11:LINE(205,50)-(255,0),11:LINE(205,145)-(25
5,191),11:PAINT(220,70),11
2400 LINE(190,165)-(220,165),6:LINE(190,165)-(193,
160),6:LINE(217,160)-(193,160),6:LINE(217,160)-(22
0,165),6:PAINT(193,162),6
2410 LINE(195,165)-(195,170),1:LINE(215,165)-(215,
170),1:CIRCLE(125,100),25,1:PAINT(125,100),1
2420 GOSUB2210:RETURN
```

*Location 4. In the misty mountains*

```
2430 REM ** location 4 **
2440 LINE(0,0)-(255,120),7,BF:LINE(0,120)-(255,191
),14,BF
2450 LINE(0,120)-(200,120),12:LINE(105,50)-(200,12
0),12:LINE(105,50)-(0,120),12:PAINT(70,119),12
2460 LINE(180,130)-(255,130),3:LINE(180,130)-(255,
15),3:PAINT(254,30),3
2470 CIRCLE(200,20),8,6:PAINT(200,20),6
2480 GOSUB2210:RETURN
```

*Location 5. Outside the cavern of light*

```
2490 REM ** location 5 **
2500 LINE(0,0)-(255,120),1,BF:CIRCLE(128,70),65,15
:PAINT(128,70),15:LINE(0,101)-(255,191),2,BF
2510 LINE(120,102)-(136,102),14:LINE(100,191)-(120
,102),14:LINE(136,102)-(155,191),14:PAINT(120,190)
,14
2520 GOSUB 2210:RETURN
```

*Location 6. In a narrow corridor*

```
2530 REM ** location 6 **
2540 LINE(0,101)-(255,191),3,BF
2550 LINE(0,0)-(255,101),4,BF:LINE(0,0)-(120,80),1
4:LINE(120,80)-(120,111),14:LINE(120,111)-(0,191),
14:PAINT(20,100),14
2560 LINE(255,0)-(155,80),14:LINE(155,80)-(155,111
),14:LINE(155,111)-(255,191),14:PAINT(190,100),14
2570 LINE(155,80)-(120,111),6,BF
2580 GOSUB 2210:RETURN
```

*Location 7. On the sea shore*

```
2590 REM ** location 7 **
2600 LINE(0,0)-(255,191),4,BF:LINE(100,100)-(200,1
00),9:LINE(80,70)-(220,70),9
2610 LINE(100,100)-(80,70),9:LINE(200,100)-(220,70
),9:PAINT(110,90),9
2620 LINE(150,70)-(152,5),1,BF:LINE(153,62)-(153,6
),13:LINE(153,6)-(200,62),13
2630 LINE(200,62)-(153,62),13:PAINT(155,50),13:LIN
E(0,100)-(255,191),7,BF
2640 LINE(210,70)-(255,80),1
2650 GOSUB2210:RETURN
```

*Location 8. In the yacht*

```
2660 REM ** location 8 **
2670 LINE(0,0)-(255,191),15,BF:CIRCLE(125,100),70,
4:LINE(55,80)-(192,80),4:PAINT(125,100),4:CIRCLE(1
25,100),73,14:PAINT(10,10),14
2680 LINE(98,78)-(120,83),3,BF
2690 GOSUB2210:RETURN
```

*Location 9. On a small island*

```
2700 REM ** location 9 **
2710 LINE(0,0)-(255,150),5,BF:CIRCLE(125,151),150,
12:LINE(0,151)-(255,191),2,BF
2720 PAINT(10,10),12:LINE(120,150)-(130,150),14:LI
NE(120,150)-(115,191),14:LINE(130,150)-(135,191),1
4:PAINT(130,190),14
2730 GOSUB2210:RETURN
```

*Location 10. On a mountain pass*

```
2740 REM ** location 10 **
2750 LINE(0,0)-(255,150),7,BF:LINE(0,151)-(255,191
),14,BF
2760 LINE(20,170)-(200,170),12:LINE(20,170)-(110,2
0),12:LINE(110,20)-(200,170),12:PAINT(110,160),12
2770 LINE(40,170)-(255,170),2:LINE(40,170)-(200,80
),2:LINE(200,80)-(255,170),2:PAINT(200,169),2
2780 LINE(5,170)-(120,170),3:LINE(5,170)-(60,70),3
:LINE(60,70)-(120,170),3:PAINT(70,165),3
2790 GOSUB2210:RETURN
```

*Location 11. In a strange room*

```
2800 REM ** location 11 **
2810 LINE(0,0)-(255,191),9,BF:LINE(120,30)-(140,14
0),1,BF
2820 LINE(140,140)-(255,191),3:LINE(120,140)-(0,19
1),3:LINE(120,30)-(0,0),2:LINE(140,30)-(255,0),2
2830 LINE(120,140)-(140,140),3:PAINT(130,150),3
2840 GOSUB2210:RETURN
```

*Location 12. In a field of corn*

```
2850 REM ** location 12 **
2860 LINE(0,0)-(255,30),7,BF:LINE(0,31)-(255,191),
3,BF
2870 LINE(20,30)-(60,20),6,BF:LINE(20,30)-(24,5),6
,BF
2880 CIRCLE(200,15),5,11:PAINT(200,15),11
2890 GOSUB2210:RETURN
```

*Location 13. Outside the church*

```
2900 REM ** location 13 **
2910 LINE(0,0)-(255,70),7,BF:LINE(0,71)-(255,191),
2,BF
2920 LINE(50,160)-(200,90),9,BF:CIRCLE(125,130),20
,1:PAINT(125,130),1
2930 LINE(105,160)-(145,140),1,BF:LINE(50,160)-(70
,50),6,BF
2940 GOSUB2210:RETURN
```

*Location 14. In the interrogation room*

```
2950 REM ** location 14 **
2960 LINE(0,0)-(255,191),1,BF:CIRCLE(125,40),15,11
:PAINT(125,40),11
```

```
2970 LINE(124,0)-(125,20),2,BF:LINE(100,140)-(150,
142),15,BF:LINE(110,140)-(112,180),15,BF:LINE(140,
140)-(142,180),15,BF
2980 GOSUB2210:RETURN
```

*Location 15. On a grassy hillside*

```
2990 REM ** location 15 **
3000 LINE(0,0)-(255,191),15,BF:CIRCLE(125,-10),180
,3:CIRCLE(125,215),80,3:PAINT(125,160),3:PAINT(0,1
91),3:PAINT(255,191),3:PAINT(125,190),3
3010 CIRCLE(215,10),20,11:PAINT(215,10),11
3020 LINE(115,140)-(130,125),4,BF:LINE(113,124)-(1
32,124),6:LINE(113,124)-(122,115),6:LINE(123,115)-
(132,124),6:PAINT(129,123),6
3030 GOSUB2210:RETURN
```

*Location 16. Outside the house*

```
3040 REM ** location 16 **
3050 LINE(0,0)-(255,191),7,BF:LINE(150,152)-(255,1
91),3,BF:LINE(150,20)-(255,152),4,BF:LINE(0,190)-(
150,152),3:PAINT(0,191),3
3060 LINE(170,80)-(200,60),15,BF:LINE(220,80)-(250
,60),15,BF:LINE(135,20)-(255,20),6:LINE(135,20)-(1
85,0),6:PAINT(255,1),6
3070 CIRCLE(200,130),10,14:PAINT(200,130),14:LINE(
190,130)-(210,152),14,BF:CIRCLE(20,30),20,11:PAINT
(20,30),11
3080 GOSUB2210:RETURN
```

*Location 17. At the end of the rainbow*

```
3090 REM ** location 17 **
3100 LINE(0,0)-(255,120),5,BF:LINE(0,121)-(255,191
),6,BF:LINE(1,191)-(110,120),14:LINE(110,120)-(140
,120),14:LINE(140,120)-(255,191),14:PAINT(125,191)
,14
3110 FOR X=1 TO 15:CIRCLE(125,190),100+X*2,X:NEXT:
CIRCLE(220,20),30,10:PAINT(220,20),10
3120 GOSUB 2210:RETURN
```

*Location 18. Next to a strange wall*

```
3130 REM ** location 18 **
3140 LINE(0,0)-(255,78),14,BF:LINE(0,79)-(255,191)
,12,BF
3150 FOR X=10 TO 250 STEP 30:CIRCLE(X,70),10,X/20:
```

```
PAINT(X,70),X/20
3160 IF X<240 THEN CIRCLE(X+15,50),10,X/20-1:PAINT
(X+15,50),X/20-1
3170 LINE(0,10+X/4)-(255,10+X/4),1:CIRCLE(X,30),10
,X/20+1:PAINT(X,30),X/20+1
3180 NEXT:GOSUB 2210:RETURN
```

*Location 19. Outside the yellow building*

```
3190 REM ** location 19 **
3200 LINE(0,0)-(255,191),7,BF:LINE(101,170)-(255,1
91),2,BF:LINE(0,50)-(100,191),10,BF
3210 LINE(0,50)-(120,50),6:LINE(120,50)-(0,0),6:PA
INT(1,1),6:LINE(70,50)-(74,20),4,BF
3220 CIRCLE(50,130),20,15:PAINT(50,130),15:LINE(30
,130)-(70,191),15,BF
3230 LINE(37,135)-(48,189),1,BF:LINE(53,135)-(63,1
89),1,BF
3240 GOSUB2210:RETURN
```

*Location 20. On a grassy plain*

```
3250 REM ** location 20 **
3260 LINE(0,0)-(255,90),5,BF:LINE(0,91)-(255,191),
14,BF
3270 LINE(10,0)-(100,191),3:PAINT(10,10),3 :LINE(1
00,100)-(170,100),12:LINE(135,10)-(100,100),12:LIN
E(135,10)-(170,100),12:PAINT(120,90),12
3280 LINE(253,0)-(155,191),9:PAINT(255,70),9
3290 GOSUB 2210:RETURN
```

*Location 21. At the entrance of a gloomy cavern*

```
3300 REM ** location 21 **
3310 LINE(0,0)-(255,130),7,BF: CIRCLE(330,100),140
,6:PAINT(255,100),6:CIRCLE(290,100),60,1:PAINT(255
,100),1:LINE(0,0)-(90,191),3:PAINT(0,191),3
3320 CIRCLE(40,20),20,12:PAINT(40,20),12
3330 GOSUB2210:RETURN
```

*Location 22. In the ghost's cavern*

```
3340 REM ** location 22 **
3350 SCREEN 2,2,0
3360 LINE(0,0)-(255,150),1,BF:LINE(0,151)-(255,191
),15,BF:LINE(0,191)-(50,151),12:LINE(0,151)-(50,15
1),12:PAINT(0,152),12
3370 LINE(200,151)-(255,151),12:LINE(200,151)-(255
,191),12:PAINT(255,161),12:RESTORE 3390
```

```
3380 FOR X=1 TO 32:READ D:S$=S$+CHR$(D):NEXT X:SPR
ITE$(1)=S$
3390 DATA 1,1,3,63,115,227,227,255,254,252,247,226
,224,224,126,31,128,128,192,252,206,199,199,255,12
7,63;247,163,3,23,190,248
3400 FOR Y=10 TO 100 STEP 10:FOR X=1 TO 255 STEP 2
:PUT SPRITE 1,(X,Y),6,1:NEXT X,Y
3410 PUT SPRITE 1,(100,100),6,1:GOSUB 2210:RETURN
```

*Location 23. In the cavern of pyramids*

```
3420 REM ** location 23 **
3430 LINE(0,0)-(255,191),1,BF:CIRCLE(125,-20),200,
11:PAINT(255,191),11:PAINT(0,191),11
3440 LINE(125,50)-(70,161),15:LINE(125,50)-(190,16
1),15:LINE(190,161)-(70,161):PAINT(125,160),15
3450 GOSUB 2210:RETURN
```

*Location 24. Outside a large office block*

```
3460 REM ** location 24 **
3470 LINE(0,50)-(255,120),14,BF:LINE(0,50)-(50,0),
7:PAINT(0,0),7:LINE(205,0)-(255,50),7:PAINT(255,0)
,7
3480 LINE(0,50)-(50,0),13:LINE(205,0)-(255,50),13:
LINE(255,50)-(0,50),13
3490 CIRCLE(125,120),20,9:PAINT(125,120),9:LINE(80
,70)-(110,90),1,BF:LINE(175,70)-(145,90),1,BF
3500 PAINT(100,0),13:LINE(0,120)-(255,191),3,BF:LI
NE(20,70)-(50,90),6,BF:LINE(220,70)-(250,90),6,BF
3510 LINE(120,120)-(130,120),1:LINE(120,120)-(70,1
91),1:LINE(130,120)-(185,191),1:PAINT(125,191),1
3520 GOSUB2210:RETURN
```

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

The background graphics for the cave are created in lines 3360-3370. Line 3380 is used to define the sprite for the ghost, whilst the data for the sprite definition is held in line 3390.

The sprite for the ghost is made to move across the screen in line 3400 until it reaches the bottom of the screen. It is then placed roughly at the centre of the screen in line 3410, before the routine to wait for a key being pressed is called.

You will notice that the subroutine at line 2210 is called at the end of each graphics subroutine. If this is left out, control will return to the main program section as soon as pictures has been drawn. This would return the program into SCREEN 0 and would, in effect, clear the graphics from view. By including the subroutine at line 2210, the player must press the space bar before control is returned to the main section of the program.

The most time-consuming part of the whole process of creating a graphical adventure is, without doubt, the graphic design stage. Further details of how to design graphics for adventure games can be found in Chapter 18, although there are one or two other approaches which can be adopted when writing a graphical adventure. In this program I have stored the coding for each location within a different subroutine. This is a method which has the advantage of drawing the pictures very quickly, but suffers the clear disadvantage of using vast quantities of memory. Other programmers prefer to store the data for their graphics within DATA lines. Doing this does tend to use less memory, although it does take far longer to develop and, in addition, it can also slow down the speed with which the pictures are drawn.

A totally different approach can be utilised if you have a disc drive for your MSX micro. The pictures for each location can be created using one of the excellent graphics packages and the whole screen can be stored on the disc. Thus a single disc can store pictures for all the different locations in the game and each time that a player enters a new location, the corresponding picture can be loaded back off the disc. Using this approach is very efficient in memory usage. No longer do you have to store all the information for the pictures within the program and therefore more space is left for extra locations and more puzzles.

Unfortunately, this approach is not really feasible for tape-based programs. With a disc drive, it is possible to search for and load a file within seconds, while the same process would take several minutes using tape. In addition, the user would have to be ready to rewind or fast forward the tape to the point where the particular picture is stored and the whole process would ruin the enjoyment to be found in playing the game.

Yet another approach is to use a light pen to aid the design of pictures for your adventure. At the time of writing, only Sanyo have actually released a light pen for MSX micros and its cost may seem rather high. The nicest thing about this light pen is that it allows the user to create superb pictures on screen and the software supplied with it creates BASIC code which can be loaded into other MSX micros not fitted with the light pen. It may seem to be an expensive approach to program development, but the results which can be achieved have to be seen to be believed. The light pen cartridge must be plugged into the cartridge port and will work on any make of MSX micro. If you've got the money to spare, you may like to try out using this accessory to help you to design the screens for your adventure. A short (52 line) program is supplied with the light pen which allows the SAVEd screens to be loaded back into an MSX micro without a light pen fitted and this routine would need to be incorporated within your own program. If you intend to sell a program designed using this system, you may need to seek permission to use this routine from Sanyo.

# 10 Snow White part 2

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 her to fend for herself in the forest. After many hours in the forest, Snow White stumbled across a small cottage, where she found seven dwarfs. 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 dwarfs 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 until this very day. You are the handsome prince who has set out on a perilous journey to revive the beautiful princess.

Before incorporating the graphics 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 KEY OFF
20 SCREEN 0:COLOR 15,4:LOCATE 13,2:PRINT"Snow Whit
e"
30 LOCATE 1,10:PRINT"An adventure for MSX microcom
puters."
40 LOCATE 8,20:PRINT"by Steve W. Lucas"
50 DIM S%(24,4),Q$(24),G$(24),B%(24),N$(24),N%(24)
,V$(4),A(24)
60 RESTORE:FOR X=1 TO 24:READ Q$(X):FOR Y=1 TO 4:R
EAD S%(X,Y):NEXT Y,X
70 REM ** data for locations **
80 DATA outside a small house. The door is   open.
 There is a footpath to the west,0,0,0,2
90 DATA on a wide road. A narrow footpath    leads
 east to a small house.,4,12,1,0
```

```
100 DATA in a small room. There's not much    furn
iture in here!,0,0,0,0
110 DATA in the misty mountains. There is a   wide
 road to the south and a narrow  footpath leads ea
st.,0,2,5,0
120 DATA outside the cavern of light. The pathlead
s into the cave.,0,0,0,4
130 DATA in a narrow corridor. To the south   lie
the mountains and I can see lightfrom the north.,7
,5,0,0
140 DATA on the sea shore. A yacht is moored  here
. The cavern of light lies to thesouth.,0,6,0,0
150 DATA in the cabin of the yacht. I can see a sm
all island in the distance.,0,0,0,0
160 DATA on a small island. The yacht is      moor
ed here. There is a large viaducthere.,0,10,0,0
170 DATA on a high mountain pass. The path is bloc
ked by a fall of rubble.,9,0,0,0
180 DATA in a strange room inside the old     chur
ch. The walls are painted red.,0,14,0,0
190 DATA in a field of ripening corn. There isa br
ightly coloured building in the  distance.,2,15,0,
13
200 DATA outside the ruins of an old church.  An o
pen doorway leads into the ruins.,0,0,12,0
210 DATA in an interrogation room. A small    tabl
e stands underneath a bright     light and a chair
 stands at one side.,11,0,0,0
220 DATA on a grassy hillside. There is a     smal
l building in the distance.,12,17,16,0
230 DATA outside a small house. The door is   lock
ed at the moment.,0,0,0,15
240 DATA at the end of the rainbow. A wide    road
 leads through the rays of light.,15,18,0,0
250 DATA next to a strange wall. It is paintedwith
 brightly coloured circles.,17,20,19,0
260 DATA outside a large yellow building. A   larg
e dog stands guard by the door.,0,0,24,18
270 DATA on a grassy plain. A green pyramid   stan
ds in the distance.,18,0,0,21
280 DATA at the entrance to a large gloomy    cave
rn.,0,22,20,0
290 DATA in a dark cavern. An evil ghost      prev
ents me moving further into the  cavern.,21,0,0,0
300 DATA in a large cavern. An enormous       pyra
mid of solid ice lies in the     centre.,0,0,0,22
310 DATA outside a large office block.,0,0,0,19
```

*Line:*

10 turns off the messages about the function keys
20 select text screen/colours and print title
30-40 titles

```
50      DIMension arrays
60      READ DATA for the locations
70-310  DATA for the locations
```

The first thing you'll notice about this listing is that I've used exactly the same variables 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$(X) | holds the description of the locations |
| G$(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 objects being carried |
| A(X) | flag to test if object is being carried |
| P% | current location |
| S% | score |

As in the previous game, each line of DATA contains a description of the location followed by four numbers. These numbers correspond to the number of the location you reach 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. I have again used the default screen width of 37 characters, although users with monitors may prefer to use 40 character width.

*Locations in 'Snow White'*

1. outside a small house. The door is open. A footpath lead 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 at 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.

```
320 FOR X=1 TO 24:READ G$(X),B%(X),N$(X):N%(X)=X:N
EXT X
330 DATA a sail,5,sail,a rope,6,rope,a rudder,7,ru
dder
340 DATA a bowl of soup,3,soup,a wild cat,4,cat
350 DATA a golden casket,23,casket,"",23,"",a gold
en harp,10,harp
360 DATA a screwdriver,9,screwdriver,a giant lizar
d,8,lizard,a brass knocker,16,knocker,"",16,""
370 DATA a large red button,24,button,an old lady,
19,lady,a pot of gold,17,gold
380 DATA an enormous man with a gun in his    hand
,14,man,"",14,""
390 DATA a pile of straw,2,straw,a wooden plank,1,
plank,a large pot hole,15,pothole
400 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, like the previous game, invisible at first and only become visible after solving some problems. These are listed in the chart below.

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

*Line:*

320 read data into the arrays.

330-400 DATA for the objects, location of objects and word recognised.

**Objects found in 'Snow White'**

| *number* | *object* | *invisible at start?* | *location* |
|---|---|---|---|
| 1. | a sail | no | 5 |

| *number* | *object* | *invisible at start?* | *location* |
|---|---|---|---|
| 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 and 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 graphical 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. (See page 108.)

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 that you move into a new location. If its value remains zero, then the program is sent to the subroutine which controls the graphics and the appropriate picture is displayed on the screen.

```
410 REM ** set variables **
420 P%=3:S%=0
430 REM ** main control loop **
440 IF K=0 THEN GOSUB 2160
450 SCREEN 0:K=0:PRINT J$
460 IF P%=4 THEN SH=SH+1:IF SH>3 THEN PRINT"The ca
t looks agitated!"
470 IF P%=4 AND SH>5 THEN E$="The cat attacks me!"
:GOSUB 1140
480 PRINT:PRINT"I am :-":PRINTQ$(P%)
490 REM ** describe directions **
500 A$="":IF S%(P%,1)>0 THEN A$="North"
510 IF S%(P%,2)>0 AND LEN(A$)>0 THEN A$=A$+",South
" ELSE IF S%(P%,2)>0 THEN A$="South"
```

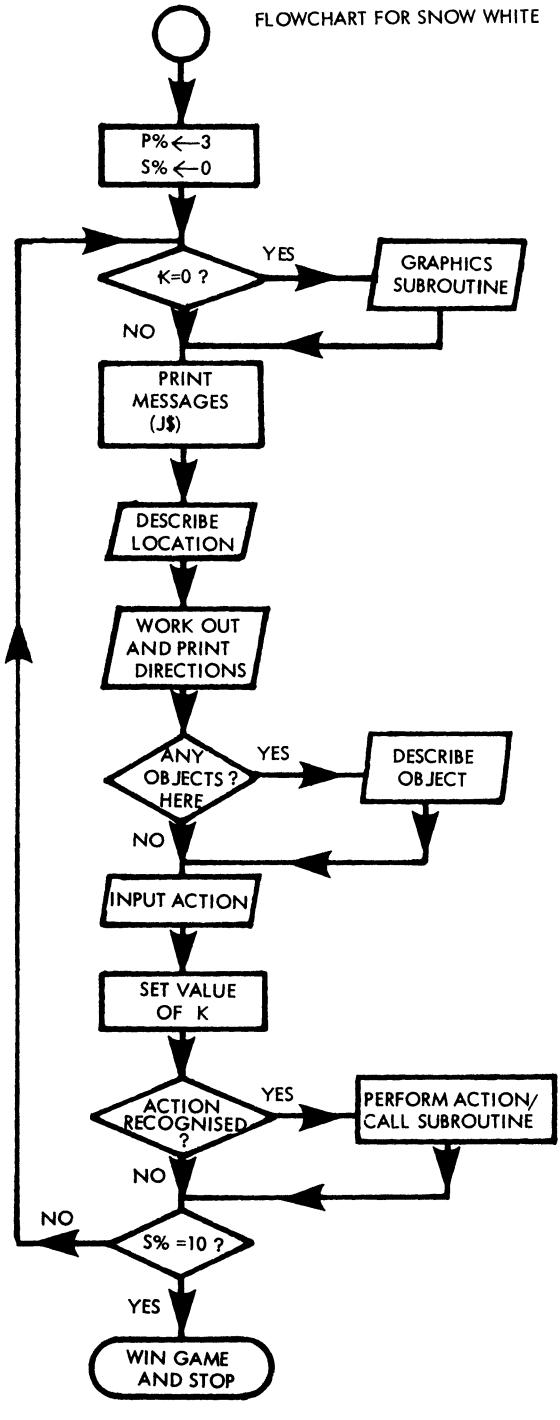



**Fig. 10.1** The flowchart for *Snow White*

```
520 IF S%(P%,3)>0 AND LEN(A$)>0 THEN A$=A$+",East"
 ELSE IF S%(P%,3)>0 THEN A$="East"
530 IF S%(P%,4)>0 AND LEN(A$)>0 THEN A$=A$+",West"
 ELSE IF S%(P%,4)>0 THEN A$="West"
540 IF P%=3 THEN A$="Out" ELSE IF P%=11 THEN A$=A$
+",Out"
550 IF P%=13 OR P%=5 OR P%=16 OR P%=1 OR P%=19 OR
P%=24 THEN A$=A$+",In"
560 IF A$="" THEN A$="nowhere obvious"
570 PRINT:PRINT"I can go :-":PRINTA$:PRINT
580 REM ** describe objects **
590 E=0:FOR T=1 TO 24
600 P=0:IF B%(T)=P% THEN P=1
610 IF P=1 THEN 630
620 NEXT T:GOTO 650
630 IF E=0 THEN PRINT"I can see :-"
640 PRINT G$(T):E=1:GOTO 620
650 PRINT:Z$="":INPUT"What should I do ";Z$
660 REM ** analyse input **
670 B$=LEFT$(Z$,2):C$=LEFT$(Z$,3):D$=LEFT$(Z$,4):J
$=""
680 BEEP:CLS:K=1
690 IF (B$="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!"
700 IF P%=15 AND (B$="s" OR D$="go s") AND SC=0 TH
EN E$="I fall down the hole and die":GOSUB 1140
710 IF (B$="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!"
720 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$="I can't go that way!"
730 IF P%=19 AND SE=0 AND (B$="w" OR D$="w") THEN
J$="The old lady won't let me pass":GOTO 440
740 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$="I can't go that way!"
750 IF C$="out" THEN GOSUB 3540
760 IF C$="in" OR D$="go i" THEN GOSUB 1050
770 IF C$="loo" THEN K=0
780 IF C$="swi" THEN GOSUB 1110
790 IF C$="get" OR C$="tak" OR C$="gra" THEN GOSUB
 1200
800 IF C$="inv" THEN GOSUB 1410
810 IF C$="dro" OR C$="lea" THEN GOSUB 1470
820 IF C$="unl" THEN GOSUB 1570
830 IF C$="pla" THEN GOSUB 1630
840 IF C$="pra" THEN GOSUB 1680
850 IF C$="kno" THEN GOSUB 1730
860 IF C$="pre" OR C$="rin" THEN GOSUB 1830
870 IF C$="rea" THEN GOSUB 1880
880 IF C$="giv" THEN GOSUB 1910
```

```
890 IF C$="sta" OR C$="kil" OR C$="use" THEN GOSUB
 1990
900 IF C$="dri" THEN GOSUB 2040
910 IF C$="sco" THEN J$="This isn't a game you kno
w!"
920 IF C$="hel" THEN J$="I'm sorry I haven't a clu
e!"
930 IF C$="sea" THEN J$="I didn't find anything!"
940 IF C$="row" OR C$="sai" OR D$="go b" THEN GOSU
B 2070
950 IF C$="lan" OR C$="dis" OR D$="go l" THEN GOSU
B 2120
960 IF C$="kis" THEN GOSUB 3580
970 REM ** if score <10 then jump back again **
980 IF S%<10 THEN GOTO 440
```

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 440 . . . . . calls subroutine at line 2160). The next line makes sure that the computer is in the text mode (SCREEN 0) and sets the value of K to zero to ensure that its value is always the same after drawing the graphics.

Immediately after the player types in his instructions, the value of K is changed to one (line 680), to suppress any graphics. Its value is changed back to zero if the player moves to a new location (lines 690-720), moves in or out of a building or types 'look' (line 770). Thus when the score is checked at line 980 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'.

In the previous adventure, SCREEN mode was not changed during the program and we were able to PRINT any messages directly onto the screen. In this game, however, the mode is changed to SCREEN 0 at the start of the loop, which would effectively clear away any messages. The easiest way to solve this problem proved to be by using the variable J$ to hold any messges. To illustrate this, consider what happens if the player types in 'help' when asked 'What should I do?' in line 650. The program will compare the input with the contents of C$ in lines 690 to 960 and will find a match in line 920. It then sets the contents of the variable J$ to hold the message 'I'm sorry I haven't a clue!' so that this can be printed immediately AFTER the screen has been cleared by the SCREEN 0 command in line 450.

*Line:*

420 set the starting position to location 3 and the score to zero.

440 check the value of K and call graphics if it has been set to zero.
450 change to text mode (SCREEN0), set the value of K and print any message held in J$.
460 test to see if the location is number 4 and increase the value of the flag SH. If SH is greater than 3 then print the message.
470 test to see if the location is number 4 and if the value of the flag SH is greater than 5, lose the game.
480 print the description for the current location.
500-530 decide if you can move north, south, east or west and include this information in A$.
540 check to see if you can move 'out' and include this information in A$.
550 check to see if you can move 'in' and include this information in A$.
560 if A$ is still empty, change the message it contains.
570 describe the directions you can move in (A$).
580-620 examine all 24 objects to see if they are in the current location.
630 if it is the first object in the location, print the message 'I can see:–'.
640 print the description of the object found and jump back to line 620.
650 empty the contents of Z$ and input the player's instructions into Z$.
670 set B$, C$ and D$ so that they contain the first few letters of the player's instructions and empty J$ of any messages.
680 BEEP to make sure that the player knows that they have pressed <RETURN>, clear the screen and set the value of the flag K to one so as to suppress any graphics.
690 deals with movement north.
700 does the player try to move south from location 15 without having dropped the plank? If so he falls down the pothole and dies.
710 deals with movement south.
720 deals with movement east.
730 does the player try to move west from location 19 without helping the old lady? If so, she won't let him pass.
740 deals with movement west.
750 does the player want to 'go out'?
760 does the player want to 'go in'?
770 if the player wants to 'look', the value of the flag K is set to zero to allow the graphics to be drawn.
780 swim?
790 call the routine to 'get' an object if the player types 'get', 'take' or 'grab'.
800 call the 'inventory' routine to see which items are being carried.

810 call the routine to 'drop' an item being carried if the player types 'drop' or 'leave'.
820 unlock?
830 play?
840 pray?
850 knock on door?
860 press or ring the bell?
870 read?
880 give?
890 stab, kill or use a weapon?
900 drink?
910 score ? . Note that the message indicates that this is not a game, even though the score is checked at the end of the loop!
920 no help is available!
930 nothing is to be found by 'searching'.
940 row, sail or 'go boat'?
950 land, disembark or leave the boat
960 kiss 'Snow White' to awaken her.
980 test the score to see if it is less than 10 and jump back to the start of the loop.

In the previous game, all of the puzzles were written inside subroutines and 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 number 4 increases the value of the flag SH (line 460). The fifth object found in the game is a wild cat and this is found in location number 4. After 3 moves in location 4, a message appears on the screen that 'The cat looks agitated!' (line 460) and after 5 moves, the cat decides to attack. When this happens, the variable E$ is set to hold the message about losing the game and the appropriate subroutine is called. This is identical to the way in which the death routine worked in the previous game.

The second puzzle is set in line 700, which tests to see if the player tries to move south from location 15 without putting the plank across the pothole first. When the plank is dropped in the correct place, the flag SC is set to one and movement south is possible. In order to solve this puzzle, the player would have had to read the description of the location very carefully. The final puzzle included in the main program control loop control occurs in line 730. A test is made of the value of the flag SE whenever you try to go west from location 19. If the player has not solved the puzzle of what the old lady wants, he will be stuck!

### Win game

```
990 REM ** win game **
1000 CLS:LOCATE 10,2:PRINT"W e l l   D o n e"
```

```
1010 LOCATE 1,10:PRINT"You have found Snow White a
nd have   kissed her. She awakes and you both  liv
e happily ever after."
1020 PLAY"l16decdecdededcd"
1030 END
```

Line 980 in the main control loop tests the value of the score (S%) and once its value reaches 10, the loop is terminated and the program reaches the win game routine at line 990. This is a very simple routine.

*Line:*
1000 clear the screen and print message.
1010 print message about what happens to 'Snow White'.
1020 play a short tune (?)
1030 end of the game.

The tune played in line 1020 is not the most amazing piece of music and you may like to experiment with this line. I have already mentioned that this game does not really have a true score. In many adventures in which the quest has a specific goal, the player can either win or lose the game and therefore the variable S% is used as yet another 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 we have already used, with only minor changes to deal with the different circumstances. It is well worth while comparing these routines with their equivalent in 'The Wizard's Quest' so as 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

```
1040 REM ** go in **
1050 IF P%=1 THEN J$="O.K.":P%=3:K=0:RETURN
1060 IF P%=5 THEN J$="O.K.":P%=6:K=0:RETURN
1070 IF P%=13 THEN J$="O.K.":P%=11:K=0:RETURN
1080 IF P%=16 OR P%=19 OR P%=24 THEN J$="The door
s locked!":RETURN
1090 J$="Don't be absurd!":RETURN
```

This subroutine is very similar to its equivalent routine in 'The Wizard's Quest'. There are three locations where the player can actually move to a different place by typing the command 'go in' and three further locations where they can reasonably be expected to try to go in, but without success. These are summarised in the chart below.

| *Location* | *go in possible?* | *new location* |
|---|---|---|
| 1. outside house | yes | 3. inside a room |
| 5. outside the cavern | yes | 6. in a corridor |
| 13. outside the church | yes | 11. in a church |

16. outside a house no
19. outside a building no
24. outside some offices no

*Line:*

1050 if you are in location 1, move to location 3, set the flag to zero and return to the main loop.
1060 if you are in location 5, move to location 6, set the flag to zero and return.
1070 if you are in location 13, move to location 11, set the flag and return.
1080 if you are in location 16, 19 or 24 change the content of the variable J$ and return.
1090 set the contents of J$ to indicate that it is not possible and return.

In lines 1050-1070, you will notice that the variable J$, which prints any messages, is set to hold the message 'O.K.' and the variable K is set to zero before returning to the main control loop. Setting K=0 has the effect of forcing the computer to print the graphics for the new location. If the player is not in the three locations tested for in line 1080, then the program will reach line 1090 and a message will be printed to indicate that the player is trying to do something which is not sensible.

**Swim**

```
1100 REM ** swim **
1110 IF P%=7 OR P%=8 THEN E$="I drown":GOSUB 1140
1120 J$="Don't be silly!":RETURN
```

In this game 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 these two locations will drown. Thus the variable E$ is set to hold the message 'I drown' and control is passed to the 'death' routine if attempting to swim in either of these locations. All the other locations are too far away from the sea and therefore when line 1120 is reached, J$ is set to hold the message 'Don't be silly!' before returning to the main loop.

*Line:*

1110 if you are in location 7 or location 8, set the variable E$ to hold the message about drowning and call the lose game routine at line 1140.
1120 set the variable J$ to contain the message 'Don't be silly' and return to the main loop.

### Lose game routine

```
1130 REM ** death routine **
1140 CLS:LOCATE 1,2:PRINT E$
1150 LOCATE 1,10:PRINT"I am dead!"
1160 LOCATE 1,20:PRINT"Press the <Space Bar> to pl
ay again."
1170 A$=INKEY$:IF A$<>" " THEN 1170
1180 RUN
```

*Line:*
1140 clears the screen and prints the contents of the variable E$.
1150 print the message about death.
1160 print the message about pressing the space bar.
1170 wait for the space bar to be pressed.
1180 RUN the program again.

This routine is used whenever the player loses the game. Before it is called, a description of the reason for losing the game must be stored in the variable E$. Note that in this game, death is the only method of losing! You may like to experiment with sound effects to accompany this routine . . . perhaps the death march!

### Get

```
1190 REM ** get objects **
1200 GOSUB 1350:IF L%<1 THEN RETURN
1210 E%=0:FOR X=1 TO 24:IF B%(X)=P% AND N%(R)=X TH
EN E%=1
1220 NEXT:IF E%=0 THEN RETURN
1230 IF R=5 OR R=10 OR R=23 THEN E$="It bites me.
Aaaaggghhhh!":GOSUB 1140
1240 IF R=16 THEN E$="The man shoots me as I try t
o touch  him!":GOSUB 1140
1250 IF R=11 OR R=13 OR R=14 THEN J$="Don't be abs
urd!":RETURN
1260 IF R=24 THEN J$="The magpie keeps pecking me!
"
1270 IF R=6 THEN J$="It's too heavy!":RETURN
1280 IF R=7 THEN J$="I can't lift her!":RETURN
1290 A(R)=1
1300 E%=0:FOR X=1 TO 4
1310 IF V$(X)="" THEN V$(X)=G$(N%(R)):E%=1:X=5
1320 NEXT:IF E%=0 THEN J$="I can't carry any more!
":RETURN
1330 B%(N%(R))=0:RETURN
```

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

*Line:*
1200 calls the subroutine to split the input sentence into two

parts and return the number of the object in the variable R.

1210-1220 search through the positions of all 24 objects to see if the object is in the current location and return to the main control loop if it isn't.

1230 death if you try to pick up object numbers 5, 10 or 23.

1240 if you try to pick up object number 16, the man shoots you.

1250 prevents you from picking up objects numbered 11, 13 or 14.

1260 if you try to get the magpie, a message is printed on the screen to give you a clue!

1270 prevents you getting object number 6.

1280 prevents you getting object number 7.

1290 set the pointer A(R) to 1 for the item being carried.

1300-1310 include object's description in the array V$(X) which is used as the 'inventory'.

1320 hands full?

1330 change pointer for the object's location to zero, which makes the object seem to disappear.

Apart from the changed line numbers, the only differences between this routine and the 'get' routine from 'The Wizard's Quest' occur in lines 1230-1280. These are all checks for items which can't be carried.

Any attempt to get the wild cat (5), the lizard (10) or the python (23), results in death in line 1230, while anyone silly enough to try carrying the man with the gun will get shot in the following line. In a similar way, you are not allowed to get the door knocker, the button or the old lady, although you won't die in the attempt. Line 1260 was inserted as a clue! When the player attempts to get the magpie, the message is printed that it keeps pecking you. If you examine the 'inventory', once you have typed the routine in, you will see that you are carrying it. If the magpie is dropped at the feet of the man with the gun, it will peck him and force him to run away! Lines 1270 and 1280 prevent you 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

```
1340 REM ** check item and split words **
1350 L$="":XX=INSTR(Z$," "):R=0
1360 L%=0:L$=RIGHT$(Z$,(LEN(Z$)-XX))
1370 IF LEN(L$)<2 THEN RETURN
1380 FOR X=1 TO 24:IF LEFT$(N$(X),LEN(L$))=L$ THEN
 L%=1:R=X
1390 NEXT:RETURN
```

Apart from the line numbers used, this routine is exactly the same as the one used in the previous program, except that the number of objects is changed to 24 in line 1380. Just to remind you: 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. If no match is found, R will remain zero.

### Inventory

```
1400 REM ** inventory **
1410 E=0:PRINT"I am carrying :-":FOR X=1 TO 4:IF V
$(X)<>"" THEN PRINTV$(X):E=1
1420 NEXT:IF E=0 THEN PRINT"not a sausage!"
1430 LOCATE 3,20:PRINT"Press <Space Bar> to contin
ue."
1440 A$=INKEY$:IF A$<>" " THEN 1440
1450 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 1430 and 1440) which require the player to press the space bar before returning to the main control loop. The reason for adding these lines is that when control returns to the main loop, the SCREEN is changed to SCREEN 0 again, which clears any message 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. Other than this slight change, the routine is identical to the previous one and you would be advised to turn back for an explanation if you are not sure how it works.

### Drop

```
1460 REM ** drop **
1470 GOSUB 1350:IF L%<1 THEN J$="I don't have a "+
L$:RETURN
1480 E%=0:FOR X=1 TO 4
1490 IF V$(X)=G$(N%(R)) THEN V$(X)="":E%=1
1500 NEXT:IF E%=0 THEN J$="I'm not carrying a "+L$
:RETURN
1510 B%(N%(R))=P%
1520 A(R)=0
1530 IF R=24 AND P%=14 THEN B%(24)=0:B%(16)=0:J$="
The magpie pecks the man and he runs away leaving
something on the ground.":N$(17)="key":G$(17)="a l
arge brass key"
1540 IF R=19 AND P%=15 THEN SC=1:Q$(15)=Q$(15)+"
    There's a plank across the hole.":B%(19)=0
1550 RETURN
```

This routine is called from the main loop whenever the player tries to 'drop' or 'leave' an object being carried. It is, again, very similar to the routine in the previous listing.

*Line:*

1470 calls the subroutine which splits the sentence into two words and returns the number of the object which you want to drop.

1480-1490 search through all four elements of the array V$(X) to see if the object mentioned is being carried and remove it from the array if it is. The flag is also set if the object can be dropped.

1500 check the value of the flag to see if it is not being carried.

1510 set the pointer for the object to the current location.

1520 set the contents of the array A(R) to zero so that the computer knows that the object is no longer being carried.

1530 check if object number 24 is dropped in location number 14 and solve the puzzle.

1540 check if object number 19 is dropped in location 15 and solve the puzzle.

1550 return to the main program control loop.

There are two puzzles in this game which are solved by dropping objects in specific locations. In line 1530, 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 magpie is dropped, the pointer B%(X) for objects numbered 24 (magpie) and 16 (man) are set to zero, so that they 'disappear' from the description of the objects found in location 14. At the same time, the description for object number 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 1540, a check is made to see if the plank, object number 19 is dropped in location 15. If it is, then it covers 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 from the normal objects by changing the pointer for its location 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

```
1560 REM ** unlock **
1570 IF A(17)=0 THEN J$="what with?":RETURN
1580 IF P%=16 OR P%=19 OR P%=24 THEN J$="The key d
oesn't fit!":RETURN
```

```
1590 IF P%<>23 THEN J$="What a ridiculous idea!":R
ETURN
1600 J$="The key turns and the casket opens!"
1610 G$(7)="Snow White":N$(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!

*Lines:*

1570 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.

1580 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.

1590 check that the player is in location 23. If not, set J$ to hold the message and return to the main loop.

1600 set the message held in J$ to tell the player that the casket opens.

1610 set the description of object number 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 910), 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 and might like to try changing line 910 to:

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

## Play

```
1620 REM ** play **
1630 IF A(8)=0 THEN J$="I can't do that yet!":RETU
RN
1640 PRINT"O.K.":PLAY"18fdefdedecdefdedec":J$="Was
n't that good eh?"
1650 IF P%=19 THEN J$="The old lady thanks me for
playing   for her and says 'to get rid of the  gho
st you must pray in the old church":SA=1
1660 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 written a very simple tune using the music macro language. You may like to try elaborating on the sound to make it sound more like a harp.

Once you have 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:*

1630 check to make sure that you are carrying the harp. If A(8) is zero, then the variable J$ is set to hold the message 'I can't do that yet!'

1640 print the message, play the tune using the music macro language facility, set the contents of J$.

1650 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.

1660 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. This can be done in a similar way to that adopted in the last game and 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

```
1670 REM ** pray **
1680 J$="O.K.":PLAY"cde"
1690 IF P%<>11 THEN RETURN
1700 IF SA=0 THEN J$="nothing happens":RETURN
1710 J$="A voice booms out 'To kill the ghost you
must use a silver sword":SB=1:RETURN
```

*Line:*

1680 set the contents of J$ to hold the message 'O.K.' and play a few notes.

1690 if praying in any location other than the church (location 11), return to main program control loop.

1700 if flag SA is zero, nothing happens and control is returned to main loop.

1710 the contents of J$ are changed to new message to tell you how to kill 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 trying to pray in the church and unless the first problem has been solved, nothing happens!

### Knock

```
1720 REM ** knock **
1730 IF P%<>16 THEN J$="There isn't much point in
doing that here!":RETURN
1740 PRINT"O.K.":BEEP:FOR X=1 TO 300:NEXT X
1750 IF SA=0 THEN J$="Nobody answers.":RETURN
1760 BEEP:PRINT"I hear someone walking towards the
   door":FOR X=1 TO 300:NEXT X
```

```
1770 IF SD=0 THEN PRINT"A man answers the door and
 throws a  silver sword onto the floor. 'Take  th
at', he says."
1780 IF SD=0 THEN G$(12)="a silver sword":N$(12)="
sword":LOCATE 1,20:PRINT"Press the <Space Bar> to
continue."
1790 IF SD>0 THEN E$="The man answers the door and
 says    'What you again.' as he hits me with a ba
ton.":GOSUB1140
1800 SD=1:A$=INKEY$:IF A$<>" " THEN 1800
1810 RETURN
```

*Line:*

1730 check the location to see if the player is trying to knock anywhere other than location 16 and if he is, set the contents of J$ to contain the message that there isn't much point before returning to the main control loop.
1740 print the message and play the sound effect.
1750 check the value of the flag SA and if it is zero, set the message that nobody answers before returning control to the main loop.
1760 play the second sound effect and print the second message.
1770 if it is the first time you have knocked on the door, a man answers and drops the sword.
1780 if it is the first time you have knocked on the door, the contents of G$(12) and N$(12) are changed.
1790 if it is the second time you have knocked on the door, the man answers and hits you. The lose game routine is called.
1800 set the flag SD to one and wait for the space bar to be pressed.
1810 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 you try to knock 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, then the value of the flag SA will remain zero and nobody will answer the door (in line 1750). If it is the first time you attempt to knock on the door, the flag SD will be zero and the description of object number 12 will be changed to make the sword appear (line 1780). At the same time, a message is printed (line 1770) 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 the sword has been thrown out, the flag SD is set to one in line 1800. If you then try to knock on the door again, the value of SD will be greater than zero and this will be trapped in line 1790, where the message about the man hitting you will be stored in E$, before calling the death routine.

Line 1800 then waits for the space bar to be pressed before returning to the main loop. The reason for this is that messages have been printed on the screen which will be erased on return to the main control loop.

### Ring

```
1820 REM ** ring **
1830 IF P%<>24 THEN J$="I can't do that here!":RET
URN
1840 PRINT"O.K.":PLAY"f":FOR X=1 TO 300:NEXT X :PL
AY"c"
1850 PRINT"I hear somebody coming.":FOR X=1 TO 300
:NEXT X
1860 E$="What do you want', says a voice from behi
nd the door. A bucket of boilingoil is thrown onto
 me from above.":GOSUB 1140
```

There is a large red button on the door in location 24 of this game, which controls the door bell, and the player will probably try to press it. The line in the main loop which calls this subroutine (line 860) responds to 'press' or 'ring'.

*Line:*

1830 if the location is not number 24, set J$ to hold an appropriate message and return to the main program.
1840 print message and play sound effect.
1850 print message about somebody coming.
1860 lose game.

This routine is used 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, namely introducing a time delay. This adds realism to the game. The bell is rung in line 1840 and a message, 'I hear somebody coming' is printed in the next line. This is followed by a short time delay, again in line 1850, before the bucket of boiling oil is flung over you. You may like to try changing the sound effect in line 1840, or even try changing the result of pressing the bell.

### Read

```
1870 REM ** read **
1880 IF A(22)=0 THEN J$="I have nothing to read!":
RETURN
```

```
1890 J$="There is a simple message written on the
paper. 'You must find the key'":RETURN
```

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

*Line:*

1880 if the piece of paper is not being carried, set the variable J$ to hold an appropriate message and return to the main loop of the program.

1890 set J$ to hold the message and return to 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 1890, the player must be carrying the paper and so the player is told that he must find the key.

### Give

```
1900 REM ** give **
1910 GOSUB 1350:IF R<>15 THEN J$="There isn't much
 point!":RETURN
1920 IF P%<>19 THEN J$="There's no point in doing
that here!":RETURN
1930 IF SA=0 THEN J$="'You haven't played the harp
 for me yet!' , she says.":RETURN
1940 IF A(15)=0 THEN J$="I don't have it!":RETURN
1950 J$="The old lady takes it from me and    runs
 away singing 'Somewhere over    the rainbow!'"
1960 B%(14)=0:B%(15)=0:FOR X=1 TO 4:IF V$(X)=G$(15
) THEN V$(X)=""
1970 NEXT:SE=1:RETURN
```

If you attempt to move west from location number 19 without having first given the pot of gold to the old lady, she will refuse to let you (see line 730). The flag used to check whether the pot of gold has been given is SE and its value must be greater than zero if you are to escape.

*Line:*

1910 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.

1920 test to see if the player is in location 19 and return to the main loop if not.

1930 test the value of SA and if it is still zero, set J$ to hold a message about playing the harp first.

1940 test to see if you are carrying the pot of gold.

1950 set the message about the old lady going.

1960 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).
1970 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, while 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, he will be stuck and I haven't included a routine which allows him to quit the game. This could make a short project for you to add to the game.

**Stab**

```
1980 REM ** stab **
1990 IF A(12)=0 THEN J$="I have no weapon suitable
!":RETURN
2000 IF P%<>22 THEN J$="Don't be so violent here!"
:RETURN
2010 J$="I kill the ghost with the silver      swor
d."
2020 Q$(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:

1995 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:*
1990 check to see if you are carrying the sword and print the message if you aren't.
2000 check to see if you are in location 22, where the ghost is to be found. Print the message and return if you are in the wrong room.
2010 set the contents of the message string.
2020 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 2020, 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 takes you 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:

2015 SK=1

3550 IF SK=1 THEN RETURN ELSE SCREEN 2,2,0

**Drink**

```
2030 REM ** drink **
2040 IF A(4)=0 THEN J$="I have nothing to drink.":
RETURN
2050 E$="I drink the soup and fall into a      deep
 stupor. It must be poisoned!":GOSUB 1140
```

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 number 4.

*Line:*

2040 check to see if the soup is being carried and return to the main loop if it isn't.

2050 set the contents of E$ to hold a message about the cause of death and call the death routine.

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 plank. This could be achieved in the following way:

1. Change line 2050 to:

2050 J$="I feel much stronger now!": SL=1:RETURN

2. Add line 1225 to the 'get routine'

1225 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 getting object 8, the harp, or object 19, the plank of wood, without SL being equal to one, then the message 'I feel too weak to lift it!' will be printed and control returned to the main program.

**Sail**

```
2060 REM ** sail **
2070 IF P%<7 OR P%>9 THEN J$="Just how am I suppos
ed to do that    here?":RETURN
```

```
2080 IF P%=8 THEN J$="I'm already sailing in the b
oat!":RETURN
2090  P%=8:SG=SG+1:K=0:IF SG>1 THEN SG=0
2100 RETURN
```

In this game, there are two locations where the player must sail the boat and these are numbered 7 and 9.

*Line:*
2070 test to see if location is numbered less than 7 or greater than 9 and print the message if it is.
2080 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.
2090 change location to number 8 and change the value of the flag SG before returning to the main program control loop.

Line 2090 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 the value 1 and in location 7 if SG is equal to zero.

Whenever the player types 'sail', 'row' or 'go boat', line 940 calls this routine dealing with sailing the boat.

### Land boat

```
2110 REM ** land boat **
2120 IF P%<>8 THEN J$="not here!":RETURN
2130 K=0:IF SG=1 THEN P%=9 ELSE P%=7
2140 RETURN
2150 END
```

This routine is called from line 950 in the main control loop by the player typing 'land', 'disembark' or 'go land'.

*Line:*
2120 if the location is not number 8, print message and return to the main program.
2130 set the flag K to zero and change position.
2140 return to the main loop.

Line 2130 is the most important line in this routine. The value of the flag K is set to zero, so as 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

```
3530 REM ** go out **
3540 IF P%=3 THEN P%=1:K=0:J$="O.K.":RETURN
3550 IF P%=11 THEN P%=13:K=0:J$="O.K.":RETURN
3560 J$="Don't be a silly billy!":RETURN
```

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

*Lines:*

3540 if 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 program loop.

3550 if the location is number 11, move to location 13, set the flag K to zero and return to main loop.

3560 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 3560 and the message 'Don't be a silly billy!' will be stored in J$ for printing on return to the main program loop.

### Kiss

```
3570 REM ** kiss **
3580 IF P%<>23 THEN J$="not here!":RETURN
3590 IF S%<9 THEN J$="The casket's locked!":RETURN
3600 J$="I kiss Snow white and she awakes!":S%=10:
RETURN
```

This routine 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:*

3580 if the location is wrong, print the message and return to the main program.

3590 if the score is less than 9, the casket must be locked so the message is printed and control is returned.

3600 print message, set score and return to 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 scores are given for solving particular problems, rather than using a separate flag.

You should now have typed all the subroutines for 'Snow White' into your computer and be ready to play the game. Do remember to save a copy before running it, so that if disaster strikes, you won't lose all your hard work. Like the previous game, you would be well

advised to check each routine out as you type it in, rather than saving your checking until the end of the game.

**Suggested improvements**
There is over 11K of memory free in a 64K MSX micro after typing this program in and this is more than adequate 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 typing in the following line:

```
905 IF C$="qui" THEN PRINT"Goodbye. Thank you for playing.":
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. The other major facility missing is a save game routine. 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

**A Journey Through Space**

This adventure game loads into your computer in two parts. The first part is the main program which controls the action of the game, while 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 and 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.

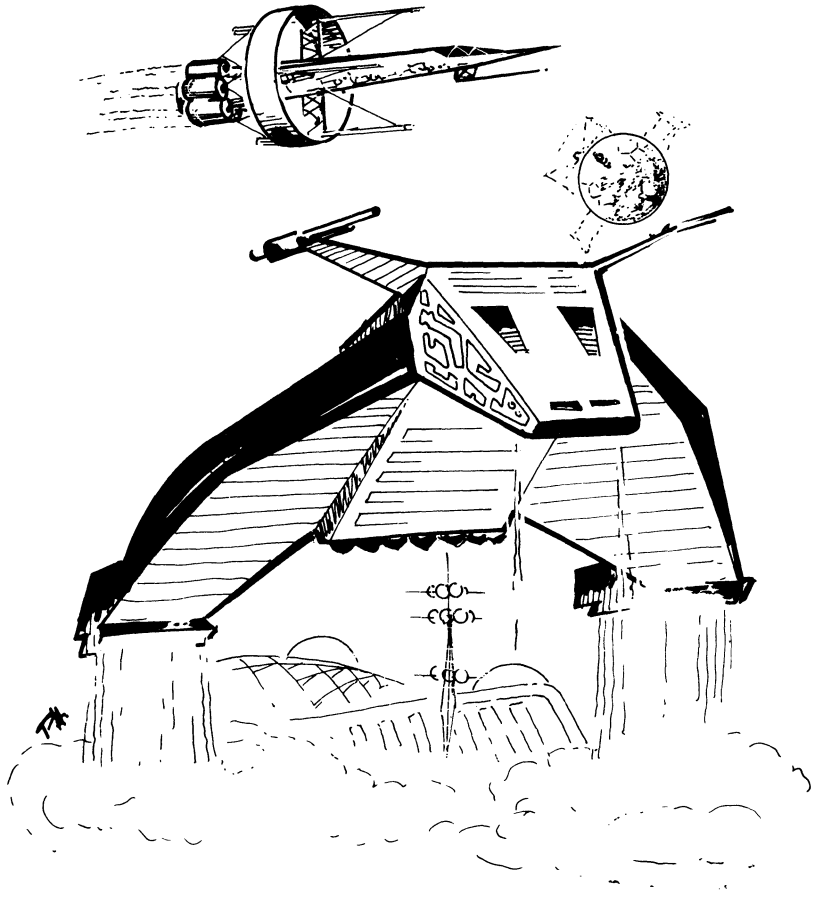
For many years, your spaceship travelled silently through galaxies far from earth, controlled only by a large computer. You, along with your fellow crew members, have remained in a state of suspended animation, your vital life functions being constantly monitored by the computer. Two hours ago, the computer 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!

*Note:*
The map for this game is to be found in chapter 15.

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 CLEAR 7000
20 DIM Q$(50),S%(50,4),V$(4),G$(25),B%(25),N%(25),
N$(25),A(25)
30 REM ** main program **
40 KEY OFF
50 SCREEN 0
60 COLOR 4,11
70 GOSUB 1880
80 LOCATE 10,2:PRINTJ$
90 LOCATE 2,7:PRINT"An adventure game for MSX micr
os"
100 CLS:S%=0:P%=2
```

*Line:*
10 clears enough string space for the data.
20 dimension the arrays.
40 turn off the messages for the function keys.
50 select text screen with default screen width.
60 select blue letters on a yellow background.
70 calls subroutine to load the data file.
80 print the title.
90 print message.
100 clear screen, set score and starting position.

The most important line in this section is line 70, which calls the subroutine to load the data file. Before this can be loaded, we must have CLEARed enough memory for the data to fit into the computer and have DIMensioned the arrays. 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 will have been read off the tape, so as to allow us to change the game without having to change any of the main program. The game always starts at location 2, although the position is loaded in off the tape so that the 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 number 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 number 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**

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.

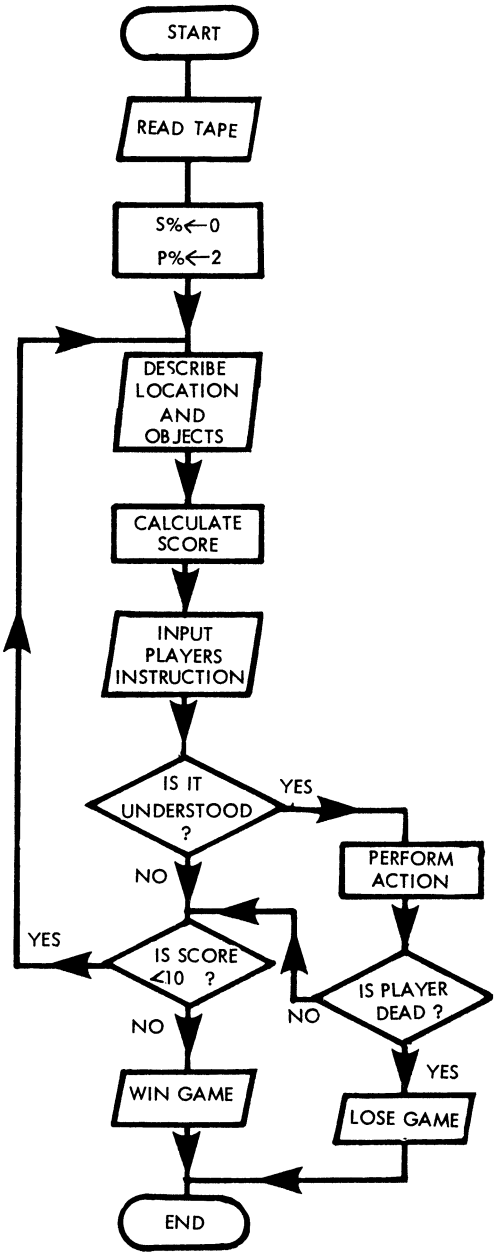



**Fig. 13.1** Flowchart for control section

```
110 PRINT"You are ":PRINTQ$(P%):PRINT
120 GOSUB 1510:REM ** check score **
130 A$="":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$="nowhere obvious!"
220 PRINT"You can travel ":PRINTA$
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 PRINTG$(T);E=1:GOTO 270
300 Z$="":PRINT:PRINT"What do you want to do now "
:INPUT Z$
310 CLS:PLAY"l32c"
320 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 IF S%<10 THEN GOTO 110
590 CLS:LOCATE 1,10:PRINT"Well Done you have solve
d this       adventure!":PLAY"116fdcfdcdedcde":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 the 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 one.
260 if 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 so as 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 than the maximum value of the loop. eg.

```
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 which can be adopted 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 350 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, then the value held in that location of the array corresponding 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 and, if they are recognised as a valid word, the appropriate subroutine is called. The list below contains all the words recognised in this section.

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 same 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.eg.

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 one 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. eg.

570 IF C$="loa" 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). eg.

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 than 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 they will be told that they have 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.

*Objects which can't be picked up*

| *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(12) is set to 2 when the player is wearing the gloves, object number 12.

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 one 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 only allowed to carry 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 so as 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$(X)<>"" THEN PRINTV$(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 one.
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 breathe the atmosphere and die in agony!!!":GOSUB 1330

A(4) would be set to zero again by dropping the space suit, while 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%(3
8,3)=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 then remove its description from V$(X) and set E=1.
900 if E is still zero, you are 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 you are 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.

# 14 A Journey Through Space

**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 number 4 or object 1, 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, while movement from location 12 takes you to location 11.

*Line:*

1020 if the player is in location 6 and the flag A(18) is still zero, you can't get past the boulder and control is returned to the main loop.

1030 if the player is 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 one. Instead of introducing yet another flag, I decided to use this and change its value to one when you fire the phaser 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 you are in location 9, move to location 6, print the message and return to the main loop.

1080 if you are 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 mentioned 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 number 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 two 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 O.K. 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 is too far bel
ow you!":RETURN
1180 PRINT"Don't be silly !":RETURN
```

There is only place in this game where this instruction works, location 34.

*Line:*

1160 check to see if the player is in location 34, move him/her to location 35 and return to the main loop.

1170 if the player is 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:RETURN
1210 IF P%=7 THEN P%=5:RETURN
1220 PRINT"not here!":RETURN
```

*Line:*
1200 if you are in location 35, move to location 34 and return to the main loop.
1210 if you are 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 him to go up into the spaceship from location 7 to location 5, while he 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 you are in location 5, move to location 7, print the message and return to the main loop.
1250 if you are 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$="Woosh! The airlock opens
and you die in the poisonous atmosphere!":GOSUB 13
40
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 number 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 number 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 space ship has already been mentioned and you will need to take great care when changing the data file that object number five remains something which must be pushed and that the player must be wearing object number 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:PRINTE$:LOCATE 1,10:PRINT"Press the <Spac
e Bar> for another    game."
1350 A$=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      pull
ed the self destruct lever!":GOSUB 1340
```

This routine was written to lure the unwary player to 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   "+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 number 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%+2
1530 IF SE=1 THEN S%=S%+2
1540 IF SF=1 THEN S%=S%+2
1550 IF SG=1 THEN S%=S%+2
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 one.

**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 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 number 25, and the crystal warp control, object number 8. The routine is fairly complex, which makes it a little more difficult to modify these objects in the data file, while 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 two when the player inserts the crystal warp control into the empty socket found in location number 2, the cabin of the spaceship. The score is also increased by two 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 two, 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$<>" " THEN 1770
1780 OPEN"cas:data" FOR OUTPUT AS #1
1790 FOR X=1 TO 50:PRINT#1,Q$(X):NEXT
1800 FOR X=1 TO 50:FOR Y=1 TO 4:PRINT#1,S%(X,Y):NE
XT Y,X
1810 FOR X=1 TO 25:PRINT#1,G$(X):NEXT
1820 FOR X=1 TO 25:PRINT#1,B%(X):NEXT
1830 FOR X=1 TO 25:PRINT#1,N$(X):NEXT
1840 FOR X=1 TO 25:PRINT#1,N%(X):NEXT
1850 FOR X=1 TO 25:PRINT#1,A(X):NEXT
1860 FOR X=1 TO 4:PRINT#1,V$(X):NEXT
1870 PRINT#1,SA,SB,SC,SD,SE,SF,SG,SH,P%
1880 CLOSE: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 4 objects being carried.
1870 save the flags SA-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 and set ready   to play"
```

```
1910 OPEN"cas:data" FOR INPUT AS #1
1920 FOR X=1 TO 50:INPUT#1,Q$(X):NEXT
1930 FOR X=1 TO 50:FOR Y=1 TO 4:INPUT#1,S%(X,Y):NE
XT Y,X
1940 FOR X=1 TO 25:INPUT#1,G$(X):NEXT
1950 FOR X=1 TO 25:INPUT#1,B%(X):NEXT
1960 FOR X=1 TO 25:INPUT#1,N$(X):NEXT
1970 FOR X=1 TO 25:INPUT#1,N%(X):NEXT
1980 FOR X=1 TO 25:INPUT#1,A(X):NEXT
1990 FOR X=1 TO 4:INPUT#1,V$(X):NEXT
2000 INPUT#1,SA,SB,SC,SD,SE,SF,SG,SH,P%
2010 CLOSE: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 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

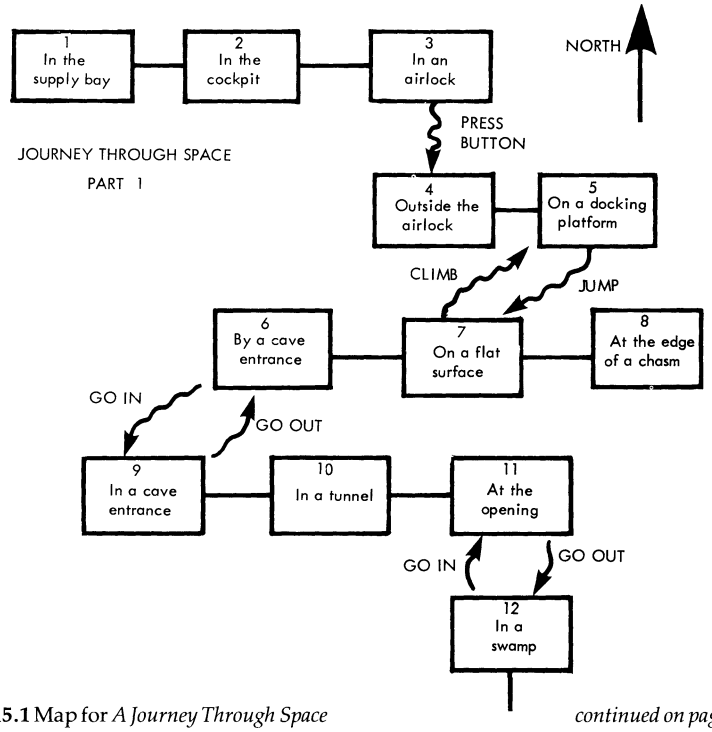
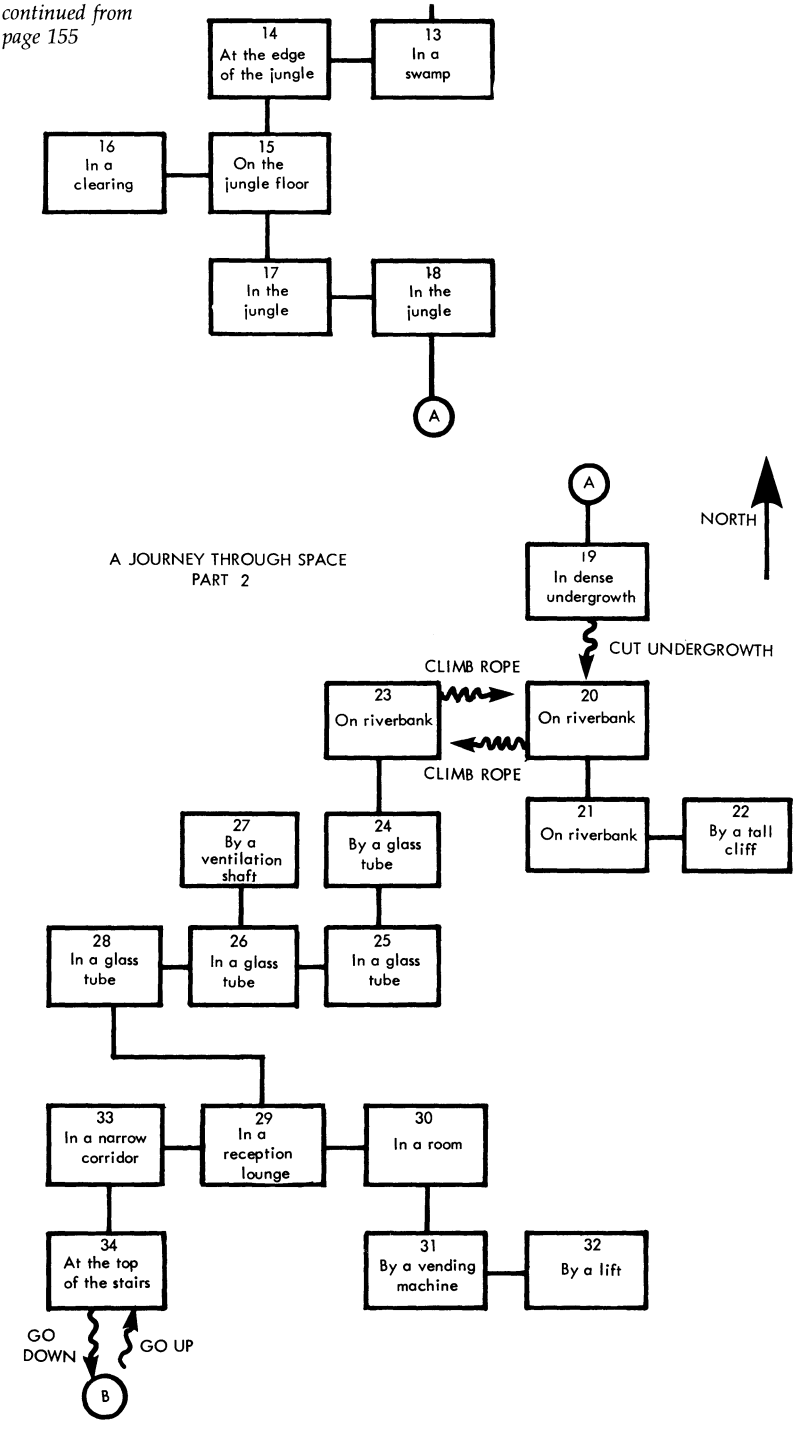
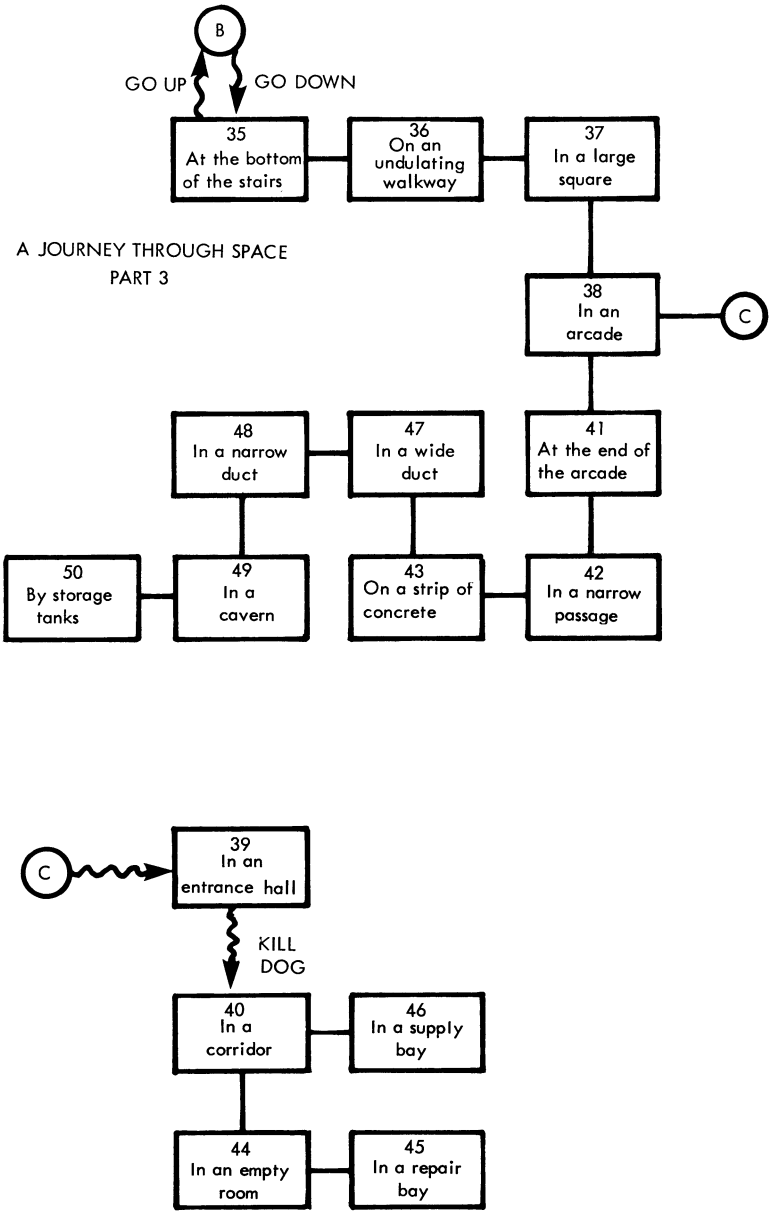



**Fig. 15.1** Map for *A Journey Through Space*

*continued on page 156*

*continued from page 155*






Careful study of this map will show you that there are fifty 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 KEY OFF
40 REM ** read data into the arrays for 'A Journey
 through Space' **
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 plan
et surface.,0,0,0,4
130 DATA outside a cave entrance.,0,0,7,0
140 DATA on the flat surface of the planet    Luci
a. The spaceship is here.,0,0,8,6
150 DATA at the edge of a deep chasm. Travel  to t
he east is impossible.,0,0,0,7
160 DATA inside a gloomy cavern. A dark tunnellead
s east.,0,0,10,0
170 DATA in a dark tunnel. Drips of water keepfall
ing on my head.,0,0,11,9
180 DATA at the cave entrance. I can see a    feti
d swamp in the distance.,0,0,0,10
190 DATA on a narrow footpath leading through the
fetid swamp. Swirls of purple    mist rise from th
e swamp. A cave      can be seen here.,0,13,0,0
200 DATA at the edge of a fetid swamp. A dry  foot
path 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 craw
l over my feet.,14,17,0,16
230 DATA in a clearing. Thick undergrowth     prev
ents me going any further west.,0,0,15,0
240 DATA on a muddy trail leading through a   dens
e jungle,15,0,18,0
250 DATA on a muddy path. The trees are alive with
 strange creatures.,0,19,0,17
260 DATA at the end of a narrow path. It looksas i
f nobody has been this way for a long time because
 the undergrowth is so dense that I can't travel f
urther south.,18,0,0,0
270 DATA on the banks of a narrow river. A    rope
 stretches across to the far side,19,21,0,0
280 DATA on the banks of a fast flowing river.A hi
gh cliff towers above me.,20,0,22,0
290 DATA underneath a tall cliff. A cave      entr
ance can be seen above my head.,0,0,0,21
300 DATA on the banks of a river of mercury. Arope
 stretches across to the far side,0,24,0,0
```

```
310 DATA on a narrow path leading into a clearglas
s 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  plan
et surface. A door leads north.,27,0,25,28
340 DATA in a ventilation shaft. It is too    narr
ow to go further north.,0,26,0,0
350 DATA in a wide glass tube leading into the top
 of a large building.,0,29,26,0
360 DATA in the reception lounge of the 'LuciaMini
ng Corporation',28,0,30,33
370 DATA in a small room full of chairs made  of a
 purple fur like material.,0,31,0,29
380 DATA by a vending machine. The two slots  are
covered with a red notice writtenin a strange lang
uage.,30,0,32,0
390 DATA by a lift. The doors are closed and  the
lights 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  line
d with plants whith eyes which   follow my every m
ove.,0,34,29,0
410 DATA at the top of a flight of stairs. A  plas
tic android stops me going down.,33,0,0,0
420 DATA at the bottom of a flight of stairs. A pa
ssage 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 building.,37,41,0,0
460 DATA inside an entrance hall. A mad LucianRock
 Hound spits molten gold at me.,0,0,0,38
470 DATA in a narrow corridor. Doors lead     sout
h and east.,39,44,46,0
480 DATA at the end of the arcade. A path     lead
s south between two buildings.,38,42,0,0
490 DATA in a narrow passage between tall     buil
dings.,41,0,0,43
500 DATA on a narrow strip of concrete at the edge
 of a sheer drop.,47,0,42,0
510 DATA in a small empty room. A door leads  east
.,40,0,45,0
520 DATA in a repair bay. It is full of tools.,0,0
,0,44
530 DATA in a supply bay. A robot stands at   the
counter 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    tank
s.,48,0,0,50
570 DATA in the fuel storage bay. A large fueldisp
enser 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  pan
el,2,panel,a panel 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 lodoria plant,18,plant,an alien mask,24,ma
sk,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.
30 turns off the messages about the function keys.
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 they are found in, the word they are recognised by 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$=INKEY$:IF AA$="N" OR AA$="n" THEN GOSUB 73
0:PRINT"DATA FILE SAVED NOW!":END
710 IF AA$="Y" OR AA$="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 he wants to save the standard game, 'A Journey Through Space', or whether he wants to write his 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 set ready
 to record"
750 PRINT:PRINT"Press <Space Bar> when ready."
760 A$=INKEY$:IF A$<>" " THEN 760
770 OPEN"cas:data" FOR OUTPUT AS #1
```

```
780 REM ** change this line to PRINT #1,J$ if you
want to save the name of the game as well...you wi
ll need to add the same line to the main game as w
ell ! **
790 FOR X=1 TO 50:PRINT#1,Q$(X):NEXT
800 FOR X=1 TO 50:FOR Y=1 TO 4:PRINT#1,S%(X,Y):NEX
T Y,X
810 FOR X=1 TO 25:PRINT#1,G$(X):NEXT
820 FOR X=1 TO 25:PRINT#1,B%(X):NEXT
830 FOR X=1 TO 25:PRINT#1,N$(X):NEXT
840 FOR X=1 TO 25:PRINT#1,N%(X):NEXT
850 FOR X=1 TO 25:PRINT#1,A(X):NEXT
860 FOR X=1 TO 4:PRINT#1,V$(X):NEXT
870 PRINT#1,SA,SB,SC,SD,SE,SF,SG,SH,P%
880 CLOSE: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-SH and the current location.
880 close the file and return.

MSX BASIC 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-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$ are printed (line 80 of the main program). 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 #1,J$

and insert the following lines into the main program.
1785 PRINT #1,J$
1915 INPUT #1,J$

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

**Changing the data**

```
890 REM ** change data for 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:PRINTQ$(X)
940 LOCATE 1,10:PRINT"What is the new description
"
950 INPUT Q$(X)
960 PRINT"Is this correct <Y>es/<N>o ?
970 AA$=INKEY$:IF AA$="n" OR AA$="N" THEN 910
980 IF AA$="y" OR AA$="Y" THEN 990 ELSE 970
990 NEXT X
1000 CLS: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:PRINTG$(X)
1040 LOCATE 1,10:PRINT"What is the new description
 "
1050 INPUT G$(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$:IF AA$="n" OR AA$="N" THEN GOTO 10
10
1090 IF AA$="y" OR 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 <RETURN>, 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 they are recognised by.

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? Cutting with a tortoise would certainly be illogical!

*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 clear the screen and 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 it is recognised by and input the word recognised.
1070 print message to ask if it is O.K.
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 number* | *description* | *word recognised* |
|---|---|---|
| 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 and 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 and 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 that 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 MSX BASIC, there are few changes needed within the program, as the PRINT #1 and CLOSE commands will operate identically in a disc or tape environment. The open command will need to be changed to take into account the fact that we are using a disc file rather than a tape file. On a disc based system, it is possible to open more than one file at any one time. Whenever the CLOSE command is issued without a channel number, it will close all the files. The open command will need to be changed to something like OPEN "A:data.DAT" FOR OUTPUT AS #1. The whole topic of sequential access files on MSX micros is more complex than simple file handling on tape, due to the

numerous types of files available. If you do have a disc drive, then you can spend many happy hours exploring the different facilities which are opened up by the disc medium. As you will probably have realised, the actual changes needed to allow this program to work from disc are minimal, although there are many new possibilities as well. One of the first things you should do is to change the messages about inserting a tape into the recorder.

# 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**

MSX machines have ten function keys available for use and so far we have completely ignored them. Moreover, the messages associated with their definitions have been turned off at the start of the games listings. Having recently been playing one of the adventure games from 'Level Nine Computing' on my Toshiba MSX machine and seeing how useful these keys can be to an adventure, I see no reason at all why we shouldn't crib a few ideas from the experts. All of the MSX versions of their games are loaded 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 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 1,"go north"+CHR$(13)
20 KEY 2,"go south"+CHR$(13)
30 KEY 3,"go east"+CHR$(13)
40 KEY 4,"go west"+CHR$(13)
50 KEY 5,"inventory"+CHR$(13)
60 KEY 6,"help"+CHR$(13)
70 KEY 7,"score"+CHR$(13)
80 KEY 8,"search"+CHR$(13)
90 KEY 9,"pray"+CHR$(13)
```

```
100 KEY 10,"wait"+CHR$(13)
110 SCREEN 0
120 LOCATE 10,2:PRINT"The Wizard's Quest"
130 CLOAD"Wizard",R
```

*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 function keys to print some of the commonest 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 short 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. You may also like to delete the line in the main program which turns off the function key messages at the bottom of the screen.

If you 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 detailed descriptions of

objects and locations such as 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 per cent 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 famous 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 number 32, movement south to location number 33, movement east is not possible, movement west takes you to location

number 41, movement up takes you to location 21 and movement down to location 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 also 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 MSX 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 has a large number of adventures available for most of the major home computers. In the MSX versions, the function

keys have been defined to contain some of the most useful commands.

*Colossal Adventure:* a superb 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 is one of my own favourites 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. Of all the games produced by Level Nine, this is my own favourite and is one I'm sure you'll enjoy too.

*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. Fortunately, the character set has not been changed in this game and I found it more enjoyable to play than Return to Eden.

*Melbourne House*

*The Hobbit:* originally written for the Sinclair Spectrum, this must be the most famous graphics adventure yet. The graphics in the MSX

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.

**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 than 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 alongside it and give it a number. The diagram on page 173 shows you how I normally cope with a game where normal logic is not obeyed.

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 it 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

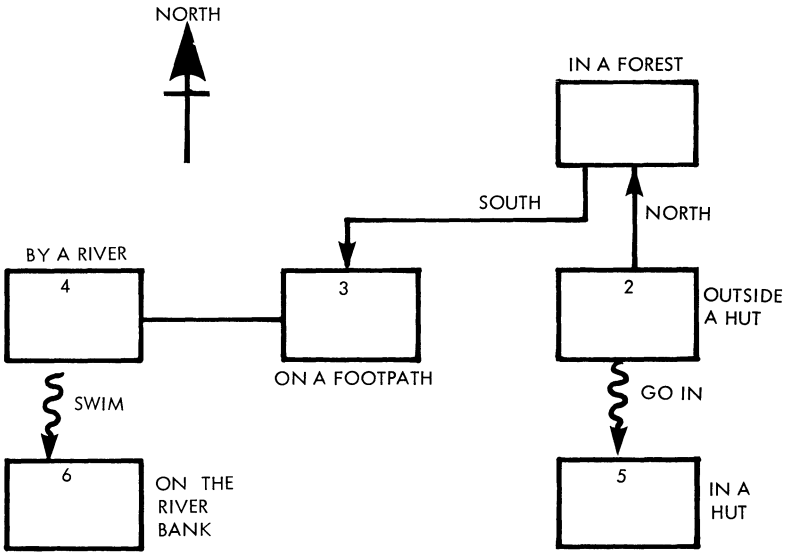



**Fig. 16.1** Constructing a map to help solve an adventure game

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 above, the player can move north from location number 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, its 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 progammer 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. It is for this reason that Melbourne House, Infocom, Adventure International, Epic and Level 9 adventures have proved so popular with enthusiasts. Quite a few companies do not, as yet, produce MSX versions of their games and if that sounds like a plea to their hearts, then you are quite right . . . . it is!

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 instructions 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 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.

# Getting to grips with BASIC 17

The three adventure listings contained within the pages of 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 the 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 and these are illustrated in the list below.

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 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 usd 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 efficent 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 this short 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 the '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.

# 18 MSX graphics

MSX machines boast some of the best graphics around. As you'll already know, there are two graphics modes to choose from: high and low resolution. The quality of picture possible in the high resolution mode (SCREEN 2) makes this the obvious choice for graphics adventures. The screen is made up of 256 pixels across by 192 down in SCREEN 2 and although it is possible to POKE into each of the screen locations using the VPOKE command, it is far easier to make use of the sophisticated commands available in the Graphics Macro Language (GML).

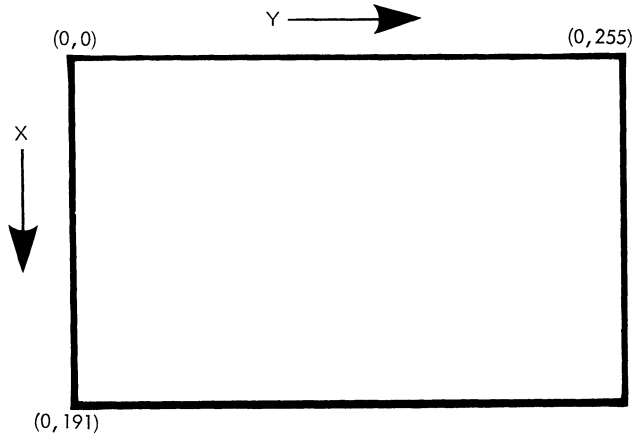



**Fig. 18.1** Screen layout in SCREEN 2

## Printing text on the graphics screen

When you first enter the graphics mode, the colour of all pixels is set to the background colour and it is important to remember that you cannot print text onto the graphics screen using the normal PRINT command. If you do want to mix text with graphics, you will need to open a channel immediately on entering the graphics mode and

close the channel again on exit. The following listing illustrates this technique:

```
10 SCREEN 2
20 OPEN "GRP:" FOR OUTPUT AS #1
.
.
.
100 PRESET (100,100):PRINT #1,"Press the Space Bar"
110 A$=INKEY$:IF A$<>" " THEN 110
120 CLOSE #1
130 SCREEN 0
```

Line 20 of the listing is used to open channel 1 to allow text to be printed on the screen and line 100 illustrates how this can be done. The cursor is moved to location (100,100) before the text is printed using the PRINT #1 command. There are two important points to note when moving the cursor to the point where you want the printing to start. The first, and most important, thing to make a note of is the colour of the screen at that point. Using the PRESET command will turn the pixel off, whereas you may like to set the colour to the foreground colour using the PSET command. In which case, line 100 would become:

```
100 PSET(100,100),3:PRINT #1,"Press the space bar"
```

The number 3 would select colour 3 (light green)

| Number | Colour |
|---|---|
| 0 | transparent |
| 1 | black |
| 2 | medium green |
| 3 | pale green |
| 4 | dark blue |
| 5 | pale blue |
| 6 | dark red |
| 7 | cyan |
| 8 | medium red |
| 9 | pale red |
| 10 | dark yellow |
| 11 | pale yellow |
| 12 | dark green |
| 13 | magenta |
| 14 | grey |
| 15 | white |

The second thing to notice is that the screen width allows only 32 characters to be displayed on each line.

Line 110 is used to prevent the program from returning to the text mode (SCREEN 0) until the space bar is pressed. If this line is left out, the program will return to text immediately after the picture has been drawn.

## PSET and PRESET

The PSET and PRESET commands are the simplest of the graphics commands and allow the colour of individual pixels to be selected. The following listing illustrates its use:

```
10 FOR X=1 TO 255
20 PSET(X,10),2
30 NEXT X
```

which will draw a green line across the screen. Each pixel's position is given by two coordinates (across,down). Whilst the PSET command is used to turn a pixel on, the PRESET command is used to turn if off again.

Although it is possible to create superb pictures using just these two commands, you'll need a lot of patience to work out the exact coordinates and with so many easy to use commands in the Graphics Macro Language this is not really the best way of creating the pictures to accompany your adventure.

## Graphics using the line command

The LINE command in MSX BASIC is one of the most versatile and useful instructions to the adventure programmer. In its simplest form, it allows straight lines to be drawn between the two coordinates specified. Try the following short program to see how this works.

```
10 SCREEN 2
20 FOR Y=1 TO 10
30 LINE (10, Y)-(100,Y),Y
40 NEXT Y
50 GOTO 50
```

Line 30 draws a line from the first coordinate to the second coordinate in the colour specified by the variable Y. It is possible to use the LINE command to draw a box by adding a simple extension to the instruction. Try the following two listings to see how this is done.

*Listing 1*

```
5 REM drawing a box
10 SCREEN 2
20 LINE(10,10)-(100,100),14,B
30 GOTO 30
```

*Listing 2*

```
5 REM draw and fill a box
10 SCREEN 2
```

```
20 LINE(10,10)-(100,100),14,BF
30 GOTO 30
```

Adding the letter 'B' to the end of the LINE instruction has the effect of drawing a box in the colour specified between the two coordinates, whilst the letters 'BF' draw and fill the box with the chosen colour.

In most of the locations in the graphics adventure in this book, I have used the LINE command at the start of each subroutine to draw the ground and the sky. The colours in MSX BASIC are all rather bright and the most suitable colours for most pictures will be 7 (cyan) for the sky and 3 (pale green) for the ground.

Listing 3 illustrates how you can draw the background (sky and ground) for your picture.

*Listing 3*

```
10 SCREEN 2
20 LINE(0,0)-(255,100),7,BF
30 LINE(0,101)-(255,191),3,BF
.
.
.
1000 GOTO 1000
```

Line 20 is used here to draw the sky, whilst line 30 draws the green grass. On some TV sets, you may find the green chosen a little pale and could try changing the colour to a darker green.

Before going further with the design of your pictures, it's worth while sitting down with a piece of graph paper or, even better, a screen layout pad from your manual and drawing a sketch of the picture you want to create. In that way you should achieve a better idea of just what effect you are trying to achieve. Do remember, though, that making the design very complicated will take a lot of programming and this in turn will mean less room for the rest of your game.

The LINE command comes in very useful when drawing buildings, mountains and any other objects with straight edges. Listing 4 illustrates how a house can be built up out of rectangles for the doors and windows and a triangle for the roof.

*Listing 4*

```
10 REM ** a house **
20 SCREEN 2
30 LINE(0,0)-(255,100),7,BF
40 LINE(0,101)-(255,191),3,BF
50 LINE(100,100)-(200,170).14.BF
60 LINE(90,100)-(210,100),10
70 LINE(90,100)-(150,65),10
80 LINE(150,65)-(210,100),10
90 PAINT(150,99),10
```

```
100 LINE(120,170)-(130,140),1,BF
.
.
.
.
1000 GOTO 1000
```

Lines 60 to 80 are used to draw the outline of a triangle for the roof in colour 10 (dark yellow) and this is filled in using the PAINT command in line 90.

Circles and arcs are also easy to draw in MSX BASIC. The simplest use of the CIRCLE command is to draw a circle, which can be filled in using the PAINT command. This is very useful for drawing the sun, moon etc in your pictures and is illustrated in listing five.

*Listing 5*

```
10 REM ** drawing circles **
20 SCREEN 2
30 CIRCLE(100,100),30,15
40 PAINT(100,100),15
50 GOTO 50
```

In this form, the CIRCLE command is followed by the coordinate of its centre (100,100), the radius (30) and the colour (15). If you want to change the aspect ratio of the circle so as to make its width greater than the height, for example, you must change line 30 in the above listing to something like:

```
30 CIRCLE(100,100),30,15,,,1/3
```

Notice that the extra commas between the colour and the aspect ratio MUST be included. The effect of this would be to draw an ellipse which is wider than it is tall. Changing the aspect ratio to 2/3 would have the opposite effect.

The final thing you can do with the CIRCLE command is to draw an arc. This is achieved by giving the computer a start angle and an end angle in radians.

The exact format of the CIRCLE command is best understood by trying a few examples. Try changing line 30 in the above listing to:

```
30 CIRCLE(100,120),50,4,3
```

or

```
30 CIRCLE(100,120),50,3,2
```

Drawing an arc and then using the PAINT command to fill in an area of the screen is a very useful method of drawing such things as hills, cave entrances, railway arches etc. The only point worth noting is that the PAINT command must be used to fill in a completely closed shape of the same colour. If you attempt to PAINT a shape which is not completely closed or which has been

drawn in a different colour, it will result in PAINTing the whole screen in the chosen colour.

The POINT command is often used in arcade games as a means of testing the colour of a pixel at a specified point on the screen. In an adventure game, however, this instruction will be of little use unless, that is, your game contains an arcade element. The DRAW command, on the other hand, is one of the most powerful and sophisticated graphics commands available. With careful use, you should be able to build up some amazing pictures using the DRAW command alone.

Drawing instructions are presented as character strings containing GML (graphics macro language) commands. Try the following example:

```
10 SCREEN 2
20 DRAW"R120D120L120U120"
.
.
1000 GOTO 1000
```

The effect produced on the screen will depend on whether or not you have previously used any of the graphics modes. If you run the program immediately after turning the computer on, a box will be drawn at the top left hand corner of the screen, but if the computer has previously used a graphics mode, the box will be drawn from the last pixel addressed. The computer has, in effect, drawn a series of lines: first right by 120 pixels then down by 120 pixels, left by 120 pixels and finally up by 120 pixels – back to where it started. In addition to L,R,U,D, there are a further four elements of GML notation dealing with movement. The full list is shown below.

*GML commands*

| | |
|---|---|
| L <n> | left by n pixels |
| R <n> | right |
| U <n> | up |
| D <n> | down |
| E <n> | move diagonally up and right |
| H <n> | move diagonally up and left |
| F <n> | move diagonally down and right |
| H <n> | move diagonally down and left |

There will be many occasions when drawing pictures for your adventure where you need to move your cursor without drawing. This can be done by using the M <x,y> command. Try the following example to see how it works.

```
10 SCREEN 2
20 DRAW"M120,99R100D120L45U23"
30 GOTO 30
```

As you can see, the computer moves the curser 120 pixels across and 99 down before starting to draw the lines. All of the above commands can be prefixed with the letters N or B. B is used to move the cursor without plotting any points, whereas N moves the cursor but returns to the place it started from when it has finished. The table below shows the full list.

A<n> sets the angle at which a line is to be drawn. It can be anywhere between 0 and 3, where 0 corresponds with 0 degrees and 3 is 270 degrees.
B moves but doesn't plot any point.
C<n> sets the colour of the line. n must correspond to the COLOR (between 0 and 15).
S<n> sets a scale factor which modifies the distance moved in the other commands.
N moves the cursor, but returns to the last point when finished.
X<string variable> allows you to execute a draw command from within a draw command. This is useful if you have previously defined a graphics shape.

The DRAW command is probably the most powerful and sophisticated method of drawing pictures on any home computer and with a bit of practice you should be able to build up some superb pictures to illustrate your game.

## Sprite graphics

Sprites are one of the most important areas of graphics programming for arcade games, but are not nearly so important when designing pictures for adventure games. They can, however, be useful to draw small characters, particularly if you choose the option in the SCREEN command to define 16 by 16 sprites. Sprites are small graphics shapes which you can define and move around the graphics screen totally independently of all other shapes on the screen. Before starting to define sprites, you really need to familiarise yourself with some of the terminology.

The graphics screen can be viewed as a plane (or surface) on which you can arrange your pictures, rather like a piece of paper. This plane is known as 'plane zero'. The sprites can be viewed as small shapes which can be placed on different planes above the screen in much the same way as sticking pieces of paper onto the background. A sprite on plane 3 will, therefore, appear to be in front of one on plane 1. Remember, though, that MSX machines will allow only one sprite at any time on any one plane.

How do we set about defining and using sprites? If you enter SCREEN 2 without any further numbers, the sprites can be defined as shapes of 8 x 8 pixels and the first thing you'll need to do is to

draw your sprite on a piece of graph paper. The diagram below illustrates this.

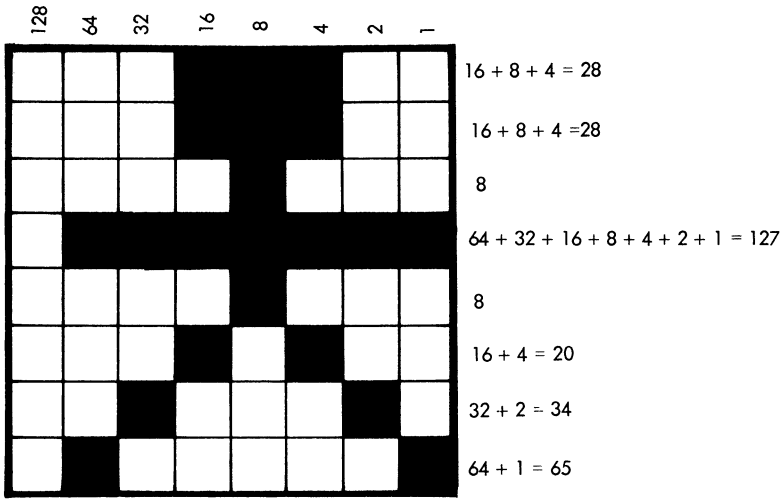



**Fig. 18.2** Sprite graphics design

Each of the small squares you've shaded in corresponds to a binary number 1 and each empty box corresponds to a binary number zero. The character is, effectively, built up out of eight rows of binary numbers and we need to convert these rows into denary (base ten). There is a way to allow the computer to do this for us, but it's not a difficult task to do for yourself and storing the BINARY numbers in DATA lines will use a lot of valuable RAM. Listing 6 illustrates how to convert the shape shown above into a sprite.

Listing 6

```
10 SCREEN 2
20 A$="":FOR X=1 TO 8
30 READ B:A$=A$+CHR$(B)
40 NEXT X
50 SPRITE$(1)=A$
60 PUT SPRITE 1,(100,100),1,1
70 GOTO 70
80 DATA 28,28,8,127,8,20,34,65
```

The SPRITE$ in line 50 is used to define a sprite. The FOR NEXT loop is used to read the eight data items corresponding to the eight rows of the character. These are then added to A$ before the SPRITE$ command converts it to a sprite. Once the sprite has been defined, it may be placed on the screen using the PUT SPRITE command, the exact format of which is described below.

PUT a SPRITE number C on PLANE number A at position (X,Y) in colour B.

```
100 PUT SPRITE A,(X,Y),B,C
```

The simple 8 x 8 sprite may be enlarged to a 16 x 16 sprite by setting an option in the SCREEN command. To do this, you should replace line 10 in the above listing with:

```
10 SCREEN 2,1,0
```

In practice, however, a more detailed sprite can be defined if you choose:

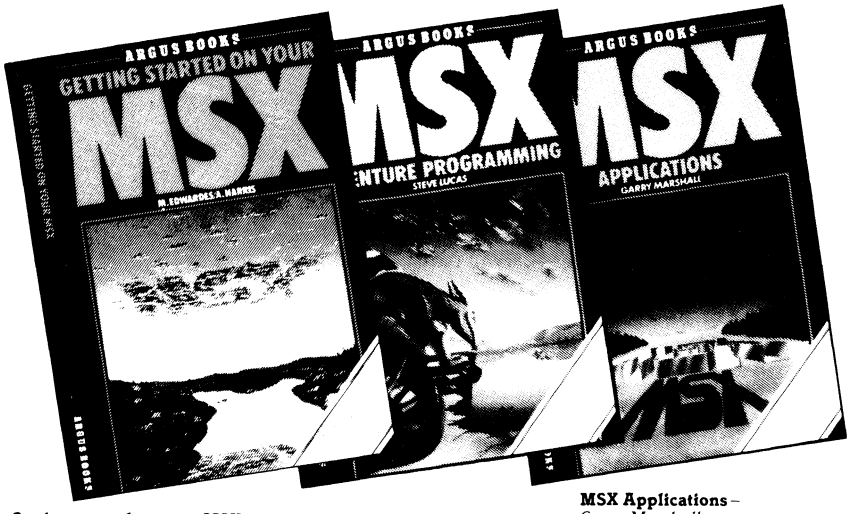
```
10 SCREEN 2,2,0
```

which allows you to define a 16 x 16 sprite from the start. You will, of course, require 32 items of DATA to deal with what are, essentially, four 8 x 8 sprites placed next to each other on the screen. The first 16 items of data deal with the left hand side of the sprite whilst the final sixteen deal with the right. The rest of the procedure is much the same as with defining 8 x 8 sprites. If you are feeling really adventurous, you can even magnify these 16 x 16 sprites to get 32 x 32 sprites using SCREEN 2,3,0.

Before designing your graphics for an adventure game, it's most important to plan the structure of your game carefully. The easiest way to do this is to enter all graphics through a short control subroutine which in turn calls the individual subroutine for each location. In that way you won't lose track of where you're up to in program development. It's all too easy to be dragged away from the keyboard for a few hours only to return and be unable to track down where you were up to. Good planning really does help you to design a program which doesn't need to be patched up to hide the flaws. With MSX BASIC you have all the tools to handle the graphics, the only limitation being the somewhat gaudy colours available.

# Index

# BOOKS FOR MSX OWNERS




**Getting started on your MSX**–
M. Edwards & A. Harris

For first time users of MSX computers. It explains all about programming in BASIC and develops a full feature BASIC arcade type game at the same time. The listing for the game is built up gradually to illustrate each new command as it is introduced. An introduction to machine code and a useful glossary of terms are also included.

Illustrated, 234 × 156mm,
192pp £6.95
0 85242 858 8
PRODUCT CODE No. 170079

**MSX Adventure Programming**–
Steve Lucas

The book to teach you how to write your own adventure programs, including developing the plot, drawing the map, and translating the objects in the game into DATA statements. High-resolution graphics and sound are also described, and listings for three typical adventure games are also included.

Illustrated, 234 × 156mm,
224pp £7.95
0 85242 857 X
PRODUCT CODE No. 170052

**MSX Applications**–
Garry Marshall

The book describes, demonstrates and illustrates the full range of useful applications for the MSX computers. From word processors, databases and spreadsheets to problem solving, from 'bolt-ons' like cassette and disc drives, printers, plotters, joysticks, light pens and mice, to communications uses–Prestel, Micronet 80, databases, private bulletin boards, MSX–net and Telecom Gold.

Illustrated, 234 × 156mm, £7.95
0 85242 852 9
PRODUCT CODE No. 170001

**Available from good bookshops and specialist outlets.**

In case of difficulties from

**ARGUS BOOKS**

P.O. Box 35 Wolsey House Wolsey Road Hemel Hempstead Herts HP2 4SS
Telephone 0442 41221

Please add 10% of the total cost ordered to cover postage and packing (minimum 50p)

**ARGUS BOOKS**

# MSX
## ADVENTURE PROGRAMMING

Here's the book to show you how to write adventure programs on your MSX micro. 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!

The second part of the book takes a close look at graphics. MSX machines boast 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 superb sound features of MSX.

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 potential MSX adventurers, here's the book to start you off.

ISBN 0-85242-857-X

9 780852 428573